2015

High performance algorithms for large scale placement problem

Tao Lin
Iowa State University

High performance algorithms for large scale placement problem

by

Tao Lin

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:

Chris Chu, Major Professor

Randall L. Geiger

Akhilesh Tyagi

Phillip H. Jones III

Ying Cai

Iowa State University

Ames, Iowa

2015

DEDICATION

I would like to dedicate this dissertation to my family without whose support I would not have been able to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research. Especially, Dr. Chu for his guidance, patience and support throughout this research. His insights have often inspired me.

ABSTRACT

Placement is one of the most important problems in electronic design automation (EDA). An inferior placement solution will not only affect the chip's performance but might also make it nonmanufacturable by producing excessive wirelength, which is beyond available routing resources. Although placement has been extensively investigated for several decades, it is still a very challenging problem mainly due to that design scale has been dramatically increased by order of magnitudes and the increasing trend seems unstoppable. In modern design, chips commonly integrate millions of gates that require over tens of metal routing layers. Besides, new manufacturing techniques bring out new requests leading to that multi-objectives should be optimized simultaneously during placement.

Our research provides high performance algorithms for placement problem. We propose (i) a high performance global placement core engine POLAR; (ii) an efficient routability-driven placer POLAR 2.0, which is an extension of POLAR to deal with routing congestion; (iii) an ultrafast global placer POLAR 3.0, which explore parallelism on POLAR and can make full use of multi-core system; (iv) some efficient triple patterning lithography (TPL) aware detailed placement algorithms.

CHAPTER 1. OVERVIEW

Very-large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. VLSI began in the 1970s when complex semiconductor and communication technologies were being developed. An electronic circuit might consist of a CPU, ROM, RAM and other glue logic. VLSI lets IC designers add all of these into one chip.

Due to the complexity of VLSI, typically, VLSI design flow is split into several stages, as shown in Fig. 1.1. The design flow starts from system specification, which describes the behavior of the target of chip. Then architectural design decides architecture of chip, e.g., RISC versus CISC and the number of ALUs. Functional design identifies the main functional unit of system and interconnect requirements between them. It also estimates the area, power and other parameters of each unit. Logical design describes the control flow, word widths, arithmetic operations, register allocation and logic operations. Based on the result of logic design, the task of circuit design is to convert the boolean expressions into a circuit representation by considering the speed and power requirements of the system. In physical design, the circuit representation is converted into a geometric representation with specific shapes on multiple layers. Based on the circuit layout obtained by physical design, the chip is ready for fabrication on a wafer. At last, each individual chip will be packaged and tested to ensure it meets all the design specifications and functions properly.

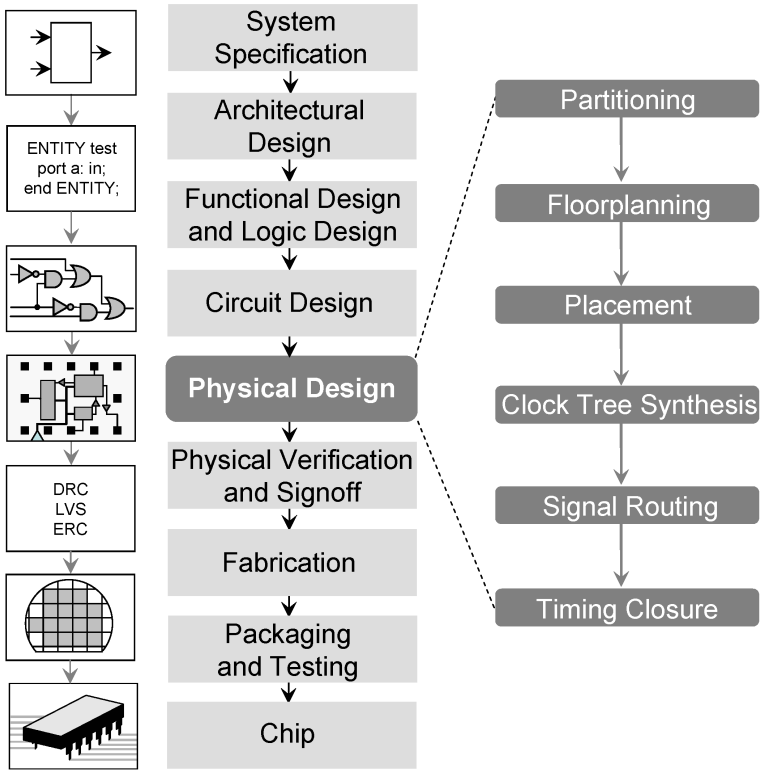


Figure 1.1 Typical VLSI design flow.

1.1 Physical Design

Physical design is one of the most important steps in VLSI design flow. As show in Fig. 1.1, it is composed of six steps: floorplanning, partition, placement, clock tree synthesis, signal routing and timing closure.

The first step is floorplanning. Floorplanning is the process of identifying structures that should be placed close together, and allocating space for them in such a manner as to meet the sometimes conflicting goals of available space and required performance. A suitable floorplan is decided based on the area of design and hierarchy. Floorplanning takes many factors into account, such as the macros used in the design, memory, other IP cores and their placement requirements, the routing possibilities and also the area of entire design. Floorplanning also decides the IO structure and aspect ratio of design. A bad floorplan will lead to waste of die area and routing congestion.

Partitioning is a process of dividing the chip into small blocks. This is done mainly to separate different functional blocks and also to make placement and routing easier. Partitioning can be done in logic design stage. The entire design is partitioned into sub-blocks and then proceeds to module design. These modules are linked together in the main module called the top level module. This kind of partitioning is commonly referred to as logical Partitioning.

Placement is performed in four optimization phases: pre-placement optimization, in placement optimization, post-placement optimization (PPO) before clock tree synthesis (CTS) and PPO after CTS. Pre-placement optimization optimizes the netlist before placement. In-placement optimization re-optimizes the logic. This can perform cell sizing, cell moving, cell bypassing, net splitting, gate duplication, buffer insertion and area recovery. Optimization performs iterations of setup fixing, incremental timing and congestion driven placement. PPO before CTS performs netlist optimization with ideal clocks. It can fix setup, hold and max trans/cap violations. It can do placement optimization based on global routing. PPO after CTS optimizes timing with propagated clock. It tries to preserve clock skew.

CTS is the process of insertion of buffers or inverters along the clock paths of VLSI design in order to achieve zero/minimum skew or balanced skew. The goal of CTS is to minimize skew and insertion delay. Apart from these, useful skew is also added in the design by means of buffers and inverters. Clock is propagated after placement because the exact physical location of cells and modules are needed for the clocks propagation which in turn impacts in dealing with accurate delay and operating frequency, and clock is propagated before routing because when compared to logical routes, clock routs are given more priority. This is because, clock is the only signal switches frequently which in acts as source for dynamic power dissipation.

The next step is signal routing. There are two types of routing in the physical design process, respectively global routing and detailed routing. Global routing allocates routing resources that are used for connections. Detailed routing assigns routes to specific metal layers and routing tracks within the global routing resources.

Finally, timing closure checks the correctness of the generated layout design. This includes verifying that the layout complies with all technology requirements: (i) design rule checking (DRC), (ii) antenna rule checking (ARC) and (iii) electrical rule checking (ERC).

1.2 Placement

Placement is an essential stage in physical design. It assigns exact locations for various circuit components within the chips core area. An inferior placement solution will not only affect the chip's performance but might also make it nonmanufacturable by producing excessive wirelength, which is beyond available routing resources. Consequently, a placer must perform the placement while optimizing a number of objectives to ensure that a circuit meets its performance requirements. Typical placement objectives include wirelength, timing, congestion and power. Minimizing the total wirelength is the primary objective of most existing placers, since wirelength minimization not only helps minimize chip size, and hence cost, but also minimizes power and delay, which are proportional to the wirelength. The clock cycle of a chip is determined by the delay of its longest path, usually referred to as the critical path. Given a performance specification, a placer must ensure that no path exists with delay exceeding the maximum specified delay. While it is necessary to minimize the total wirelength to meet the total routing resources, it is also necessary to meet the routing resources within various local regions of the chips core area. A congested region might lead to excessive routing detours, or make it impossible to complete all routes. Power minimization typically involves distributing the locations of cell components so as to reduce the overall power consumption, alleviate hot spots, and smooth temperature gradients.

Since the placement problem is very complex, to overcome the complexity issue, it is further divided into three steps: global placement, legalization and detailed placement. Global placement aims at generating a roughly legalized placement solution that may violate some placement constraints while maintaining a global view of the whole netlist. For example, overlaps among cells are allowable during global placement. Compared with the other two steps, global placement plays a key role. It has the most important impact on solution quality, and has been the focus of most prior research works. Legalization makes the rough solution from global placement legal by moving modules around locally. Detailed placement further improves the legalized placement solution in an iterative manner by rearranging a small group of modules in a local region while keeping all other modules fixed.

1.2.1 Progress and challenge of placement

Markov et al. [1] gave a good introduction on the evolution of placement techniques. Back to the 1970s, the first netlist partitioning method [2] was developed in the industry, and subsequently motivated improvements in graph partitioning heuristics. Analytical placers [3, 4, 5] started appearing in the early 1980s, but were eclipsed by combinatorial techniques when simulated annealing [6] was invented. Annealing-based placers [7, 8, 9] dominated industry use and academic results for a decade, but by the mid 1990s, annealing was no longer scalable for larger designs. Despite the steady improvement rate of analytical placement, partitioning-based methods [10, 11, 12, 13, 14, 15, 16] were improved enough to provide leading-edge performance: (i) (multilevel) Fiduccia-Mattheyses (FM) heuristics [17, 18, 19, 20, 21] produced much better results much faster than previous methods, (ii) the use of end-case techniques (optimal partitioning and end-case placement) during top-down layout optimization provided high-quality detailed placement, and (iii) the use of flat and multilevel FM heuristics was carefully optimized, including cut line selection and hierarchical whitespace allocation. By 2005, several analytical techniques have matured to the point where they reliably outperformed min-cut placement on contemporary large global placement instances.

Markov et al. [1] also made a conclusion on the topics of placement, including (i) wirelength-driven placement [7, 10, 11, 8, 9, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34], (ii) mixed-sized placement [22, 23, 24, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37] (i.e., simultaneous placement of both cells and macros), (iii) routability-driven placement [38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53], (iv) timing-driven [54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 54, 64] and power-driven placement [65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75], (v) the integration of global placement into physical synthesis [76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87], (vi) legalization [88, 89, 90, 91, 92, 93] and detailed placement [94, 95, 96, 97, 98, 99, 100, 101, 102, 103] as well as (vii) 3D placement [104, 105, 106, 107, 108, 109].

Although placement has been extensively studied for several decades, it is still a very challenging problem. Firstly, the scale of placement is increasing continuously to tens of millions cells nowadays. Secondly, placement is used in early design stages (e.g., physical synthesis)

to guide the design process, and it is typically run many times to explore the design space. Last but not least, multiple objectives, such as wirelength, timing and routability, should be optimized simultaneously, and the typical approach is to transform the problem into a sequence of wirelength-driven placement problems. Therefore, Alpert et al. [110] indicates that placement is still a hot topic.

1.2.2 State-of-the-art placement techniques

Problem scale has significant impact on the evolution of global placement core engine. In the early age, simulated annealing based placers (e.g., Timberwolf [7]) perform very well for small design. Then industry switches to min-cut based placement techniques (e.g., Capo [10] and Dragon [8]) for medium design. When design scale arrives at hundreds of thousands cells or even several millions, analytical placers [29, 11, 24, 31, 34, 22, 27, 33, 30, 32, 26] are considered state-of-the-art.

The traditional wirelength-driven global placement can be formulated as follows. A circuit can be represented by a hypergraph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_{|V|}\}$ is the set of cells and $E = \{e_1, e_2, \ldots, e_{|E|}\}$ is the set of nets. Global placement tries to determine physical positions of cells without violating placement density constraints. We denote the x-coordinates of cells by a vector $\mathbf{x} = (x_1, x_2, \cdots, x_{|V|})$, and the y-coordinates by $\mathbf{y} = (y_1, y_2, \cdots, y_{|V|})$, the objective is to minimize the HPWL which is measured by Formula 1.1 under density constraints.

$$\text{HPWL}(\mathbf{x}, \mathbf{y}) = \Sigma_{e \in E}[\max_{i \in e} x_i - \min_{i \in e} x_i + \max_{i \in e} y_i - \min_{i \in e} y_i] \tag{1.1}$$

To model density constraints, the most popular method is to split placement region into a set of uniform bins B by a $m \times n$ grid. If the placement target density λ is given, then density constraints can be formulated in Formula 1.2, where $overlap(v_i, b_j)$ is the overlap area between cell v_i and bin b_j.

$$\sum_{v_i \in V} overlap(v_i, b_j) \leq area(b_j) * \lambda, \forall b_j \in B \tag{1.2}$$

There are two branches for analytical placers, respectively quadratic placer and nonlinear placer. In quadratic placer, HPWL is approximated by a quadratic function, which is also called

quadratic wirlength. While in nonlinear placer, HPWL is usually approximated by a smooth differential function except [32]. Chu gave a good introduction on both quadratic placer and nonlinear placer in [111].

1.2.2.1 Quadratic placer

Assuming that all the nets only connect two different cells in the circuit. For any net, as shown in Formula 2.1, the HPWL is given by Manhattan distance between the two connected cells. In quadratic placer, the Manhattan distance is approximated by squared Euclidean distance of the two connected cells, so the cost function ϕ of global placement can be defined in Formula 1.3.

$$\phi = \frac{1}{2}\mathbf{x}^T Q_x \mathbf{x} + \mathbf{c}_\mathbf{x}^T \mathbf{x} + \frac{1}{2}\mathbf{y}^T Q_y \mathbf{y} + \mathbf{c}_\mathbf{y}^T \mathbf{y} + const \tag{1.3}$$

where the connection matrices Q_x and Q_y are both sparse symmetric positive definite. Minimizing ϕ is equal to solving the linear system 1.4.

$$Q_x \mathbf{x} + \mathbf{c}_\mathbf{x} + Q_y \mathbf{y} + \mathbf{c}_\mathbf{y} = 0 \tag{1.4}$$

In real circuit, many nets have more than 2 pins. Therefore, to get Formula 1.3, multiple-pin nets are decomposed into 2-pin nets by net model. There are several net models. In clique model, for a k-pin net with net weight c, the weight of each 2-pin net in the clique is usually set to either $\frac{c}{k-1}$ [112] for $\frac{2c}{k}$ [113, 11]. In star model [112, 114], one extra cell called the star cell is introduced for each net. The star cell is placed together with other movable cells during placement. Viswanathan [115] et al. proved that the clique model and the star model are equivalent in quadratic placement if the net weights are set properly. The most widely used net model is called Bound2Bound (B2B) [31] net model, which can accurately model HPWL in a quadratic placement framework. The B2B net model is based on the idea of removing all inner two-pin connections and utilizing only connections to the boundary pins. With this, the boundary pins span the net, and the property of the HPWL being the distance between the boundary pins is emulated.

Just minimizing quadratic wirelength would lead to lots of overlap among cells, to reduce cell overlap, spreading method is applied. There are many ways to generate spreading force.

Kraftwerk2 [31] and DPlace [25] are based on density gradient; Kraftwerk2 utilizes a Poisson potential by a generic supply and demand system, while DPlace models the diffusion process by solving a differential equation relating to cell density. mFAR [116] achieves the spreading forces by moving cells from those bins with overflow to those without. FastPlace [115] and RQL [23] move the cells from high density bins to the low density adjacent bins by cell shifting. SimPL-like placers, such as [27, 28, 117, 33, 42], use roughly legalized placement to guide spreading while keeping cells' relative positions.

1.2.2.2 Nonlinear placer

Another category of analytical approach is to formulate the placement problem as a single nonlinear program. Generally speaking, in nonlinear placer, both HPWL and density constraints are approximated by a smooth differential function. To approximate the wirelength, APlace [29], mPL [24], and NTUPlace [26], ePlace [34] all use the log-sum-exponential wirelength function described in a patent by Naylor et al. [118]. To smooth the density function, APlace and NTUPlace use a bell-shaped function proposed also by Naylor et al. [118], mPL uses inverse Laplace transformation, ePlace uses a new model derived from electric field. Recently, Zhu et al. proposed a non-smooth optimization method [32], which exactly models HPWL and density constraints rather than approximating them.

In nonlinear placer, density constraints are modeled as a penalty function, and then added into cost function. Since the models of HPWL and density constraints are nonlinear, the resulting nonlinear program can be solved by any nonlinear programming algorithms. In APlace and NTUPlace, the nonlinear program is converted by the quadratic penalty method into a sequence of unconstrained minimization problems, which are solved by conjugate gradient method [119]. In mPL, the nonlinear program is solved by the Uzawa algorithm [120]. in [32], the non-smooth optimization is solved by sub-gradient method [121].

In quadratic placer, quadratic wirelength is minimized by solving a sparse positive definite symmetric linear system, which is very fast and runtime complexity is almost linear. However, in nonlinear placer, the cost of solving nonlinear optimization is much higher. Therefore, to handle large-sized problems, a multilevel scheme is commonly used in nonlinear placers, such as

NTUPlace3 and mPL. First, a hierarchy of coarser netlists is constructed by clustering heavily connected modules together. Second, an initial placement of the coarsest netlist is generated. Finally, the netlist is successively unclustered, and the placement at each level is refined. The multilevel scheme can improve both the runtime and the solution quality of analytical placement algorithms. There are several popular clustering algorithms, such as First Choice clustering technique [18], Best Choice clustering technique [122] and Safe Choice clustering technique [123].

1.3 Research Contributions

This dissertation includes the following contributions:

- A high performance placement core engine, POLAR [33]. It adopts the popular framework of rough legalization [27]. An elegant and effective algorithm for look-ahead legalization is proposed. The experimental results over ISPD 2005 benchmark suite [124] and ISPD 2006 benchmark suite [125] verify that POLAR is very comparable to state-of-the-art academic placers in both runtime and placement quality.

- An efficient routability-driven placer, POLAR 2.0 [45]. It targets on mitigating routing congestion by the following two basic approaches: (1) minimizing routing demand by maintaining a good wirelength-driven placement; (2) spreading the routing demand properly by a novel routability-driven rough legalization and a history based cell inflation. Experimental results over ICCAD12 benchmark suite [51] show that POLAR 2.0 outperforms all published academic routability-driven placers.

- An ultrafast global placer, POLAR 3.0. It is based on POLAR and explore parallelism. To achieve high scalability that previous works have not reached to, the global placement iterations are divided into a series of *frame*, in which partitioning is applied base on cells' locations and then placement of each partition is performed simultaneously. Experimental results show that POLAR 3.0 can make full use of multi-core system and it delivers up to 7-30× speedup over state-of-the-art academic placers, with competitive solution quality.

- Some efficient TPL-aware detailed placement algorithms: (i) a TPL-aware detailed placement algorithm which can leverage any existing detailed placer as black-box; (ii) a TPL-aware detailed placement refinement algorithm [126], which optimizes placement perturbations and TPL objectives (such as lithography conflicts and stitch count) simultaneously.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 introduces our placement core engine POLAR. Chapter 3 talks about our routability-driven placer POLAR 2.0. Chapter 4 introduces POLAR 3.0. Chapter 5 and Chapter 6 are related to TPL-aware detailed placement algorithms. Finally, conclusions are made in Chapter 7.

CHAPTER 2. POLAR: A HIGH PERFORMANCE WIRELENGTH-DRIVEN PLACER

Wirelength is one of the most important metrics in placement problem. Minimizing wirelength is not only a beneficial, but also fundamental step to optimize other metrics, such as timing, power and routability. In this chapter, we propose a high performance mixed-size wirelengh-driven placer called POLAR. POLAR is based on the recent popular look-ahead legalization idea. The goals of our look-ahead legalization are: (i) to achieve a roughly legalized placement; (ii) to maintain cells' relative positions of quadratic placement while minimizing cell movements. To achieve these goals, in POLAR, look-ahead legalization is realized in a simple and elegant manner. Firstly, all placement density hotspots (where placement overflow occurs) are detected. Secondly, for each hotspot, an appropriate window is searched to cover it by enumerating many feasible candidates. Finally, cell-to-bin assignment is performed within each window by a fast recursive bisection method. The experimental results verify the efficiency of POLAR over the ISPD 2005 benchmarks [124] and ISPD 2006 benchmarks [125].

2.1 Introduction

Placement is one of the most fundamental problems in electronic design automation (EDA). Although it has been extensively studied and its solution quality has been improved significantly during the last decade, a high performance placer is still in urgent need to catch up with the continual increase of design scale. Besides, considering varieties of new constraints and objectives introduced due to technology scaling, Alpert et al. [110] indicates that placement is still a hot topic.

There are many metrics in placement optimization. Wirelength is one of the most important metrics, since minimizing wirelength is not only a beneficial but also fundamental step to optimize other metrics, such as timing, power and routability. Therefore, [1] points out that designing more efficient wirelength-driven placer is the key step to conquer new emerging challenges in placement problem.

To solve placement problem, analytical approach defines a suitable analytical cost function and minimizes the cost function through numerical optimization methods. It is considered the most promising technique. Depending on the cost function, analytical placers can be subdivided into the following two categories: nonlinear placer and quadratic placer.

Nonlinear placer approximates half perimeter wirelength (HPWL) by a nonlinear cost function, e.g., log-sum-exp function. And the placement density constraints are smoothed by differentiable nonlinear function, e.g., bell-shaped function and inverse Laplace transformation [24]. Because solving nonlinear programming is time consuming, nonlinear placers usually apply a multilevel approach [127, 123] to reduce runtime. Examples of nonlinear placers are APlace [29], mPL [24] and NTUPlace [26].

Different from nonlinear placer, quadratic placer approximates HPWL by a convex quadratic function, which is also called quadratic wirelength. Quadratic wirelength can be efficiently minimized by solving linear equations. However, minimizing just quadratic wirelength would lead to considerable cell overlapping. Therefore, many techniques have been proposed to spread out cells while maintaining quadratic nature of optimization.

Among cell spreading techniques for quadratic placer, iterative force-directed approach is the most promising one due to its low runtime and good placement quality. It interprets placement problem as a classical mechanics problem of finding equilibrium configuration for a spring system. In each placement iteration, the equilibrium state of the corresponding spring system is achieved by minimizing the quadratic wirelength. Then a cell spreading technique is applied to generate anchors for movable cells. Based on these anchors, additional spreading forces are added into spring system. This process gradually spreads out cells until the cell distribution is almost even and the wirelength is not improved any more. Examples of quadratic

placers which apply force-directed approach are Kraftwerk2 [31], DPlace [25], mFAR [116], FastPlace [22], RQL [23], SimPL [27], ComPLx [28] and MAPLE [117].

The main difference among different force-directed quadratic placers is how they spread out movable cells to generate their anchors. Kraftwerk2 [31] and DPlace [25] are based on density gradient. Kraftwerk2 utilizes a Poisson potential by a generic supply and demand system, while DPlace models the diffusion process by solving a differential equation relating to cell density. mFAR [116] achieves the spreading forces by moving cells from those bins with overflow to those without. FastPlace [22] and RQL [23] move the cells from high density bins to the low density adjacent bins by cell shifting. Recently, SimPL [27] proposed a new cell spreading technique called look-ahead legalization. The key idea of look-ahead legalization is that almost legal placement is used to guide the anchor generation. Many placers [27, 28, 44, 42, 117] adopt this idea and produce high quality placements. In SimPL [27], the look-ahead legalization is implemented by top-down geometric partitioning and non-linear scaling. In ComPLx [28], the entire placement process is modelled by subgradient primal-dual Lagrange optimization, whereas look-ahead legalization is modelled by a feasibility projection. In MAPLE [117], the look-ahead legalization of SimPL is combined with multilevel clustering [127] and improvement of iterative local refinement [22]. Besides, both SimPLR [44] and Ripple [42] extend SimPL to handle routing congestion.

In this chapter, we propose a new force-directed quadratic placer called POLAR. POLAR also adopts the look-ahead legalization idea and the look-ahead legalization is realized in a simple and elegant manner. We notice that while the placement solution by quadratic based wirelength minimization may have considerable overlaps, the relative positions of cells can be trusted in producing a legal placement with good wirelength. Hence, during look-ahead legalization, our goal is to maintain relative positions of cells as best as we can while minimizing cell movements. To achieve this goal, firstly, all placement density hotspots are detected. Secondly, for each hotspot, an appropriate window (which is also called expansion region) is searched to cover it by enumerating many feasible candidates. Finally, cell-to-bin assignment is performed within each window by a fast recursive bisection method and then cells within each bin are spread out.

Comparing with other SimPL-like placers (which adopt look-ahead legalization such as [27, 28, 117], there are two main differences in POLAR's look-ahead legalization approach. The first one is how to find window for placement density hotspot. POLAR enumerates many feasible windows in order to maintain quadratic placement maximally. The second one is how to perform cell spreading in each window. POLAR formulates this step into cell-to-bin assignment so that it can have better control on the placement density of each bin. The experimental results over ISPD 2005 and ISPD 2006 benchmarks show that POLAR outperforms other SimPL-like wirelength-driven placers. Besides, POLAR's look-ahead legalization approach can be easily extended to consider routing congestion in [45], which outperforms all the other academic routability-driven placers both on runtime and quality over ICCAD 2012 routability-driven placement contest benchmarks [51] so far.

The rest of this chapter is organized as follows. Section 2.2 presents the preliminary. Section 2.3 elaborates the POLAR's algorithm. Section 2.4 presents some implementation details. Section 2.5 shows the experimental results. Finally, the conclusions are made in Section 2.6.

2.2 Preliminary

A circuit can be represented by a hypergraph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_{|V|}\}$ is the set of cells and $E = \{e_1, e_2, \ldots, e_{|E|}\}$ is the set of nets. Global placement tries to determine physical positions of cells without violating placement density constraints. We denote the x-coordinates of cells by a vector $\mathbf{x} = (x_1, x_2, \cdots, x_{|V|})$, and the y-coordinates by $\mathbf{y} = (y_1, y_2, \cdots, y_{|V|})$, the objective is to minimize the HPWL, which is measured by Formula 2.1.

$$\text{HPWL}(\mathbf{x}, \mathbf{y}) = \Sigma_{e \in E}[\max_{i \in e} x_i - \min_{i \in e} x_i + \max_{i \in e} y_i - \min_{i \in e} y_i] \tag{2.1}$$

2.2.1 Quadratic optimization

Assuming that all the nets only connect two different cells in the circuit. For any net, as shown in Formula 2.1, the HPWL is given by Manhattan distance between the two connected cells. In quadratic placer, the Manhattan distance is approximated by squared Euclidean distance of the two connected cells, so the cost function ϕ of global placement can be defined

in Formula 2.2.

$$\phi = \frac{1}{2}\mathbf{x}^T Q_x \mathbf{x} + \mathbf{c}_\mathbf{x}^T \mathbf{x} + \frac{1}{2}\mathbf{y}^T Q_y \mathbf{y} + \mathbf{c}_\mathbf{y}^T \mathbf{y} + const \tag{2.2}$$

where the connection matrices Q_x and Q_y are both sparse symmetric positive definite. Minimizing ϕ is equal to solving the linear system 2.3.

$$Q_x \mathbf{x} + \mathbf{c}_\mathbf{x} + Q_y \mathbf{y} + \mathbf{c}_\mathbf{y} = 0 \tag{2.3}$$

In POLAR, preconditioned conjugate gradient (PCG) method with incomplete Cholesky decomposition [128] is used to solve linear system 2.3.

2.2.2 Bound-to-bound net model

In real circuit, lots of nets have more than two pins. To get the quadratic cost function in Formula 2.2, every multi-pin net should be decomposed into a set of 2-pin nets by a net model, e.g., clique model [11], hybrid model [22] or Bound-to-bound (B2B) model [31]. The net model determines the connection matrices Q_x and Q_y, which have big impact on the runtime and placement quality of quadratic placer. Therefore, it is important to choose suitable net model. In POLAR, we use B2B net model, which has been shown to both accurate and efficient in practice.

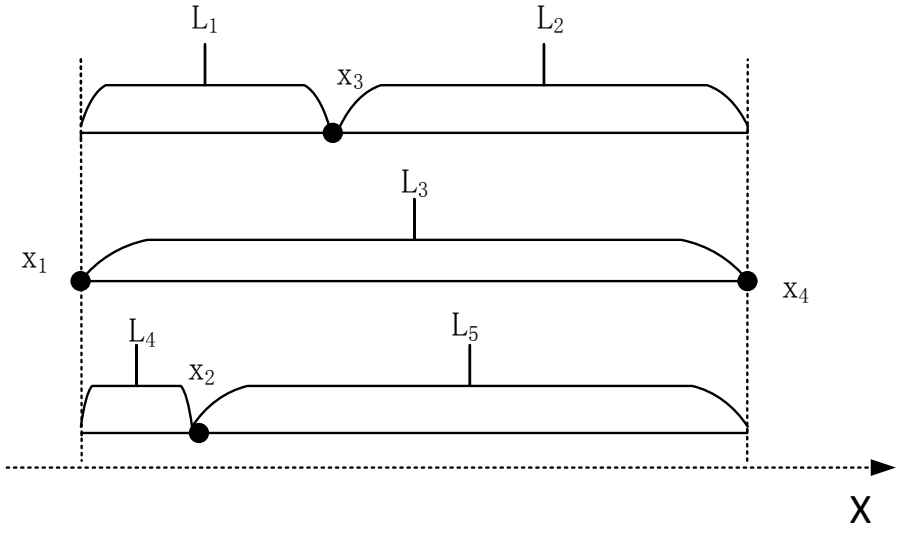


Figure 2.1 B2B net model.

The B2B net model is based on the idea of removing all inner two-pin connections and utilizing only connections to the boundary pins. With this, the boundary pins span the net, and the property of the HPWL being the distance between the boundary pins is emulated. In x-direction, the two-pin connection weight $w^x_{p,q}$ of B2B net model is determined in Formula 2.4. In y-direction, the weight is calculated in the similar way with x-direction. B2B net model should be updated once cells' positions are changed in each global placement iteration. Fig. 2.1 gives an example of B2B net model, there are four pins in the given net. The left most pin is x_1 and the right most pin is x_4. So the quadratic wirelength of this net in x-direction is $w^x_{1,3}(x_1 - x_3)^2 + w^x_{3,4}(x_3 - x_4)^2 + w^x_{1,4}(x_1 - x_4)^2 + w^x_{1,2}(x_2 - x_1)^2 + w^x_{2,4}(x_2 - x_4)^2$, where $w^x_{1,3} = \frac{1}{2L_1}$, $w^x_{3,4} = \frac{1}{2L_2}$, $w^x_{1,4} = \frac{1}{2L_3}$, $w^x_{1,2} = \frac{1}{2L_4}$ and $w^x_{2,4} = \frac{1}{2L_5}$.

$$w^x_{p,q} = \begin{cases} 0 & \text{if pin } p \text{ and } q \text{ are inner pins} \\ \frac{2}{P-1}\frac{1}{|x_p - x_q|} & \text{otherwise} \end{cases} \tag{2.4}$$

where P is the number of pins in the net.

It is proved that Formula 2.2 based on B2B net model is completely equal to Formula 2.1 if the positions of cells are finally converged using B2B net model in [31].

2.2.3 Spreading force realization

To reduce cell overlapping, spreading forces are added to guide cells toward their anchors (some papers also call them target positions). [116] proposed a simple way (which is called fixed-point technique) to add spreading force by pseudo net connecting cell's original position to its anchor. Fig. 2.2 gives an example, where the objective function is formulated as a quadratic penalty function: $\rho\left[(x_1 - x_1^{anch})^2 + (x_2 - x_2^{anch})^2 + (x_3 - x_3^{anch})^2 + (x_4 - x_4^{anch})^2\right]$, where ρ is the weight of pseudo net. In POLAR, the weight of all pseudo nets are the same. This penalty function is added to Formula 2.2. Then the connection matrices Q_x and Q_y are updated and linear system 2.3 is solved again. In POLAR, we also use the fixed-point technique.

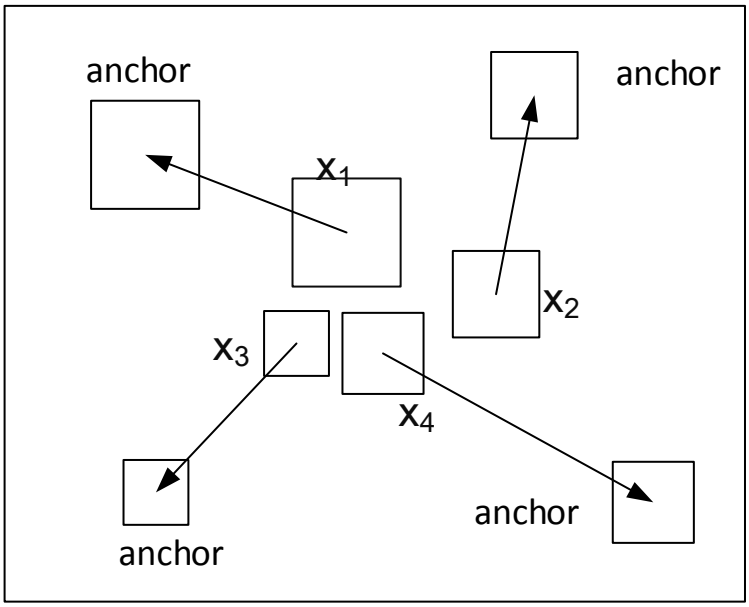


Figure 2.2 An example of the fixed-point technique.

2.3 POLAR's Algorithm

2.3.1 Algorithm outline

As shown in Fig. 2.3, POLAR is composed of three stages: initial placement, density-driven placement and post-global placement.

In the initial placement stage, a good wirelength-driven seed placement without considering cell overlapping is generated. Firstly, the hybrid net model [103] is responsible to the initial connection matrices, and linear system 2.3 is solved by PCG to get the initial placement. Next, the B2B net model updates the connection matrices to further optimize the wirelength iteration by iteration. Usually, three iterations are enough.

In the density-driven placement stage, POLAR adopts the iterative look-ahead legalization framework [27]. In each iteration, the look-ahead legalization is used to generate upper bound wirelength, and the quadratic wirelength achieved by solving linear system (2.3) is considered as lower bound wirelength[1]. To realize look-ahead legalization, firstly, all placement density

[1]This is not a real lower bound on the wirelength of the placement problem because spreading forces that are generated by heuristics are added to the linear system.

hotspots are detected. Secondly, for each hotspot, a minimal window is searched to cover it by enumerating the feasible candidates. Finally, the movable cells are evenly assigned to each bin within the window by a recursive bisection method and then the cells within each bin are spread out. After look-ahead legalization is finished, the cells' new positions are used as anchors to generate spreading forces and the connection matrices Q_x and Q_y are also updated based on anchors' positions by B2B net model [31]. The density-driven placement runs iteratively until it satisfies the convergence condition, which is defined as that the gap between lower bound wirelength and upper bound wirelength is less than 8%.

Finally, in the post-global placement stage, look-ahead legalization is applied once more. And then legalization and detailed placement are performed by FastDP [103] (using the same setting as [103]'s) to get a legal placement.

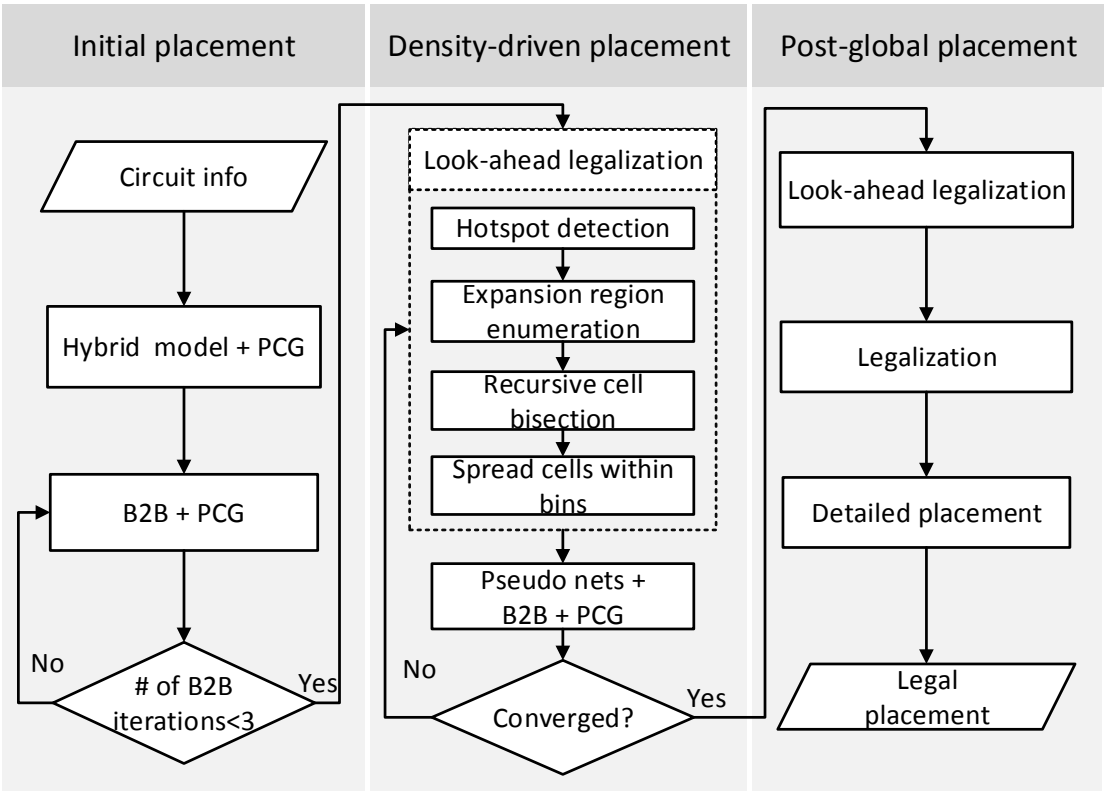


Figure 2.3 The overview of POLAR.

2.3.2 Placement density estimation

One of the goals of look-ahead legalization is to achieve a roughly legalized placement. To evaluate whether a placement is roughly legalized, a method to estimate placement density is necessary.

A popular and widely used approach is to split the placement region into a set of $p \times q$ uniform bins denoted by $B = \{b_{1,1}, b_{1,2}, \ldots, b_{1,q}, b_{2,1}, b_{2,2}, \ldots, b_{p,q}\}$. To simplify placement density estimation, we define the following concepts.

Definition 1. *For any movable cell v_i, if its geometric center is located in bin $b_{x,y}$, we say that v_i is **belonged** to $b_{x,y}$. And the placement density of bin, $d_{i,j}$, is defined as Formula 2.5, where $o_{i,j}$ and $a_{i,j}$ are the total area of movable cells belonged to $b_{i,j}$ and the available area of $b_{i,j}$ respectively. For mixed-size placement problem, $a_{i,j}$ can be calculated offline, while $o_{i,j}$ should be calculated online.*

$$d_{i,j} = \frac{o_{i,j}}{a_{i,j}} \tag{2.5}$$

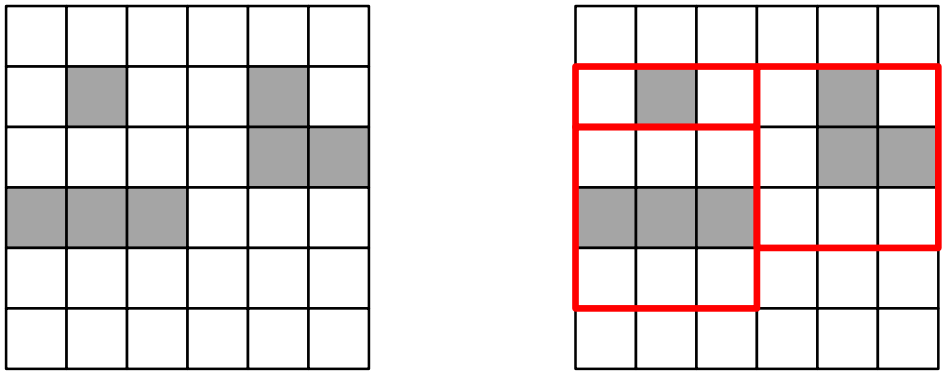
Definition 2. *A bin $b_{i,j}$ is considered density **overflow** if the following condition 2.6 is satisfied.*

$$d_{i,j} > \lambda \times \theta \tag{2.6}$$

*where λ is set to 1.05 in POLAR and θ is the density target of the circuit. Otherwise, we call bin $b_{i,j}$ **underflow**.*

The size of bin is determined as follows. Suppose the average area of standard cell is $\overline{area}$ and the expected number of standard cells in a bin is $\overline{n}$, then the bin is a square whose width (height) is calculated by Formula 2.7. In POLAR, the default value of $\overline{n}$ is 30, and the bin size is fixed during the global placement.

$$Grid_h = Grid_w = \sqrt{\frac{\overline{n} \times \overline{area}}{\theta}} \tag{2.7}$$



(a) Placement density hotspots. (b) Window cover hotspots.

Figure 2.4 Placement density hotspots and windows.

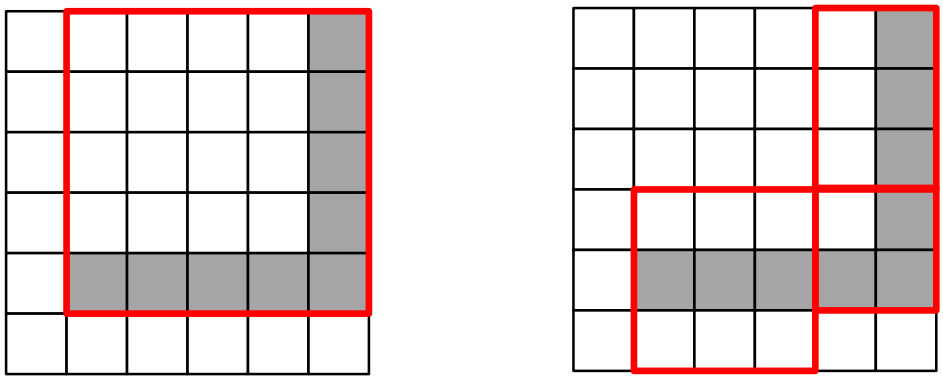
2.3.3 Hotspot detection

The first step of look-ahead legalization is to recognize placement density hotspots. Similar to SimPL [27], the placement density hotspot is defined as a cluster of overflow bins. Its formal definition is given as follows.

Definition 3. *The **grid graph** for the uniform bin grid is the graph in which (i) each bin represents a vertex, and (ii) two vertices are joined by a graph edge if and only if the two bins for those vertices are directly adjacent, either horizontally or vertically. That is, referring to a bin by its (row,column) coordinates in the uniform bin grid, bins (i, j) and (k, l) are adjacent if and only if $|k - i| + |l - j| = 1$.*

Definition 4. *A **placement density hotspot** is a spatially contiguous collection of overflow bins, i.e., a connected subgraph of overflow bins in the grid graph. A hotspot is also called a "clump." For any pair of bins in the clump, there is a path in the grid graph connecting them. The edges of the path can only be vertical or horizontal, and this path cannot go through the bins outside of the clump.*

Fig. 2.4(a) gives an example of placement density hotspots. The shadowed bins are overflow, so there are three placement density hotspots according to our definitions.



(a) Ill-shaped hotspot and corresponding window. (b) Ill-shaped hotspot decomposition and windows.

Figure 2.5 Ill-shaped hotspot and its decomposition.

The algorithm of hotspot detection is presented in Algorithm 1. Breadth first search (BFS) is used to traverse all overflow bins. Once the number of overflow bins in currently constructing density hotspot exceeds 3, a new overflow bin is set as a new start for BFS, the reason is explained in Section 2.3.4. The time complexity of Algorithm 1 is O(pq), since each bin is visited at most twice.

2.3.4 Window enumeration

To spread out the cells in placement density hotspot, we define window as follows.

Definition 5. *The **window** of a hotspot is a set of bins which completely cover the hotspot and has enough available space to accommodate the movable cells within it while satisfying the density constraints.*

As shown in Fig. 2.4(b), the three placement density hotspots in Fig. 2.4(a) are fully covered by three windows, respectively. Therefore, if the cells are evenly distributed within all those windows, a roughly legalized placement is expected.

As mentioned before, one goal of look-ahead legalization is to maintain the cells' relative positions of quadratic placement while minimizing the cell movements. Generally speaking, for any placement density hotspot, smaller window means less cell movements so it is preferred.

Algorithm 1 Placement density hotspot detection

Require: The density of bins are already calculated. The bin grid is $p \times q$. The number of bins in each hotspot is constricted to 3.
Ensure: The set of placement density hotspots, denoted by Ω.

for $i = 1 \rightarrow p$ **do**
  for $j = 1 \rightarrow q$ **do**
    visited[i][j]=0;
  end for
end for
for $i = 1 \rightarrow p$ **do**
  for $j = 1 \rightarrow q$ **do**
    if visited[i][j]==1 $\parallel$ $d_{i,j} \leq 1.05 \times \theta$ **then**
      continue;
    end if
    $H = \{b_{i,j}\}$; $count = 0$; visited[i][j]=1;
    Initialize an empty queue Q; push bin $b_{i,j}$ into Q;
    while $Q \neq \emptyset$ **do**
      pop a bin $b_{x,y}$ from Q;
      for each $b_{x,y}$'s adjacent bin $b_{x1,y1}$ **do**
        if visited[x_1][y_1]=0 && $d_{i,j} > 1.05 \times \theta$ **then**
          if $count > 3$ **then**
            $\Omega = \Omega \cup \{H\}$; break;
          end if
          $H = H \cup \{b_{x_1,y_1}\}$; $count = count + 1$;
          visited[x_1][y_1]=1; push b_{x_1,y_1} into Q;
        end if
      end for
    end while
  end for
end for

To avoid unnecessarily big window for ill-shaped hotspot, we simply constrict the number of overflow bins in each placement density hotspot, as shown in Algorithm 1. As shown in Fig. 2.5, if the number of overflow bins in each hotspot is constricted to 3, the ill-shaped hotspot is decomposed into three smaller ones.

For any placement density hotspot, there are many candidates of window. According to the definition of window, the number of candidates is $O(p^2q^2)$. Enumerating all of the candidates is time consuming. To trade off runtime and quality, we define τ-enumerated window and the minimal τ-enumerated window is chosen to spread out placement density hotspot.

Definition 6. *For any placement density hotspot, a* τ***-enumerated window*** *is a rectangular set of bins, whose geometric center is also the gravity center of corresponding placement density hotspot. Besides, its aspect ratio is within the range* $\left[\frac{1}{\tau}, \tau\right]$.

Algorithm 2 Window enumeration for a placement density hotspot

Require: $p \times q$ bins of placement region, a hotspot H, target utilization θ.
Ensure: window (l_x, l_y, r_x, r_y) for hotspot H.

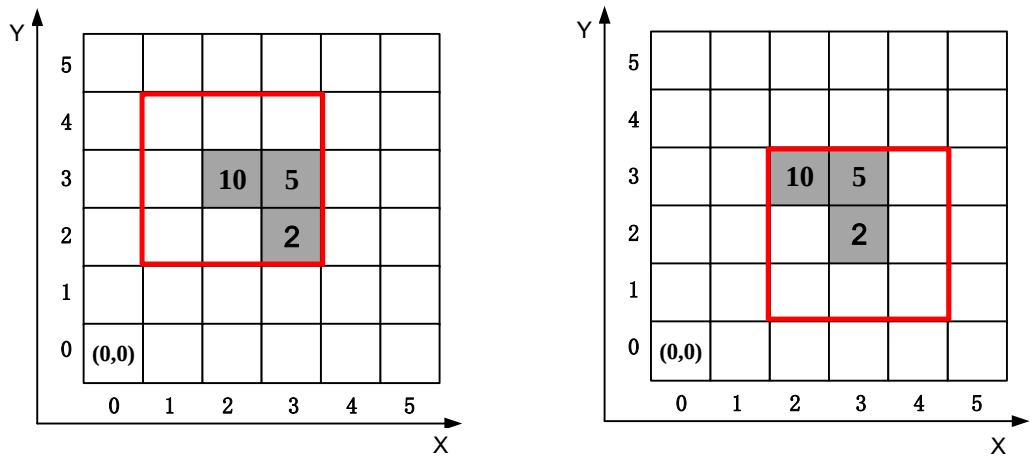
Calculate the gravity center (g_x, g_y) of H according to Formula 2.8 and 2.9;
$\Gamma = \emptyset$;
$found = false$;
$\tau = 2.5$;
while $not found$ **do**
  for $radius_x = 1 \rightarrow \max\{g_x, p - g_x\}$ **do**
    for $radius_y = 1 \rightarrow \max\{g_y, q - g_y\}$ **do**
      $l_x = \max\{0, g_x - radius_x\}$;
      $l_y = \max\{0, g_y - radius_y\}$;
      $r_x = \min\{p - 1, g_x + radius_x\}$;
      $r_y = \min\{q - 1, g_y + radius_y\}$;
      if window (l_x, l_y, r_x, r_y) does not cover H **then**
        continue;
      end if
      Calculate the space utilization ratio $\gamma = \frac{\sum_{b_{i,j} \in H} o_{i,j}}{\sum_{b_{i,j} \in H} a_{i,j}}$ of window (l_x, l_y, r_x, r_y);
      if $\gamma < \theta$ && $\frac{r_y - l_y}{r_x - l_x} \in \left[\frac{1}{\tau}, \tau\right]$ **then**
        Push window (l_x, l_y, r_x, r_y) into Γ;
        break;
      end if
    end for
  end for
  if $\Gamma = \emptyset$ **then**
    $\tau += 0.5$;
  else
    $found = true$;
  end if
end while
return the (area) minimal window from Γ;

To calculate the geometric center of window and gravity center of placement density hotspot, we use the bin coordinate system. A rectangular-shaped window can be represented by a quadruple (l_x, l_y, u_x, u_y), where (l_x, l_y) is the coordinate of its lower-left bin, and (u_x, u_y) is the coordinate of its upper-right bin. Its geometric center is defined as $\left(floor(\frac{l_x + l_y}{2}), floor(\frac{u_x + u_y}{2})\right)$. The gravity center (g_x, g_y) of placement density hotspot H is defined as Formula 2.8 and 2.9.

$$g_x = floor(\frac{\sum_{b_{i,j} \in H} o_{i,j} * i}{\sum_{b_{i,j} \in H} o_{i,j}}) \tag{2.8}$$

$$g_y = floor(\frac{\sum_{b_{i,j} \in H} o_{i,j} * j}{\sum_{b_{i,j} \in H} o_{i,j}}) \tag{2.9}$$

For instance, as shown in Fig. 2.6, the red rectangular-shaped window is represented by quadruple $(1, 2, 3, 4)$, and its geometric center is $(2, 3)$. The corresponding placement density



(a) τ-enumerated window. (b) Non τ-enumerated window.

Figure 2.6 Bin coordinate system and τ-enumerated window.

hotspot is composed of three overflow bins, the total area of movable cells belonged to those bins are 10, 5 and 2 respectively, its gravity center is $(2, 3)$ according to Formula 2.8-2.9. Therefore, the red rectangular-shaped window in Fig. 2.6(a) is a τ-enumerated window, whose aspect ratio τ is equal to 1, while the red rectangular-shaped window in Fig. 2.6(b) is not a valid τ-enumerated window, since its geometry center is not the same as the gravity center of the density hotspot. Note that, by only enumerating τ-enumerated windows, POLAR would miss some candidates of windows. However, the CPU runtime consumption is reduced significantly.

A window whose aspect ratio is either excessively high or excessively low is not beneficial to wirelength. Because the x-direction wirelength would be sacrificed if using excessively low aspect ratio window, while the y-direction wirelength would be sacrificed if using excessively high aspect ratio window. For example, as shown in Fig. 2.7, suppose that the total occupied area of each overflow bin is 9 and the total occupied area of underflow bins are all 0. If the available area of each bin is 3 and density target θ is 1, the window in Fig. 2.7(a) is expected to produce better wirelength than the window in Fig. 2.7(b). (The y-direction wirelength of Fig. 2.7(b) may be a little better than that of Fig. 2.7(a), but the x-direction wirelength of Fig. 2.7(b) may be much worse than that of Fig. 2.7(a).)

The window enumeration method is presented in Algorithm 2. The initial value of τ is set to 2.5. All the τ-enumerated windows are checked to find the minimal one. If no τ-enumerated window has enough available area to accommodate the cells within it while satisfying the density constraint, the value of τ is gradually increased by 0.5.

Algorithm 2 guarantees to return a window for the given hotspot. Its time complexity is $O(\tau pq)$ as line 15 (calculating the space utilization ratio of window) can be computed in constant time by a look-up table method, which we will show next.

In a $p \times q$ grid, for any rectangular-shaped window denoted by $(0, 0, x, y)$ whose lower-left bin is $(0, 0)$ and upper-right bin is (x, y), its available area is denoted by $z_{x,y}$. Then the available area of window (l_x, l_y, u_x, u_y) which is denoted by $a\_(l_x, l_y, u_x, u_y)$, can be calculated according to Formula 2.10.

$$a\_(l_x, l_y, u_x, u_y) = z_{u_x,u_y} - z_{l_x,u_y} - z_{u_x,l_y} + z_{l_x,l_y} \tag{2.10}$$

The same method can be applied to calculate the occupied area of windows. Therefore, if we maintain two 2-D arrays to store available area and occupied area, the space utilization γ of window can be computed in constant time. Besides, this kind of 2-D array can be constructed and updated by dynamic programming based on the following recursive relation 2.11.

$$z_{x,y} = z_{x-1,y} + z_{x,y-1} - z_{x-1,y-1} + a_{x,y} \tag{2.11}$$

Only the look-up table of occupied area should be updated once the placement is changed. Once the cell distribution of a window denoted by (l_x, l_y, u_x, u_y) is changed and the cell distribution outside of this window is untouched, the occupied area of three regions should be updated. They are respectively (1) $x \in [l_x, u_x], y \in [l_y, u_y]$, (2) $x \in [u_x + 1, p), y \in [l_y, u_y]$ and (3) $x \in [l_x, u_x], y \in [l_y + 1, q)$. Note that the boundary conditions (e.g. $l_x = 0$ or $l_y = 0$) and update order should be taken care of. The update of look-up table for occupied area is presented in Algorithm 3. Algorithm 3 should be invoked each time when a window is roughly legalized.

Algorithm 3 Update of look-up table for occupied area

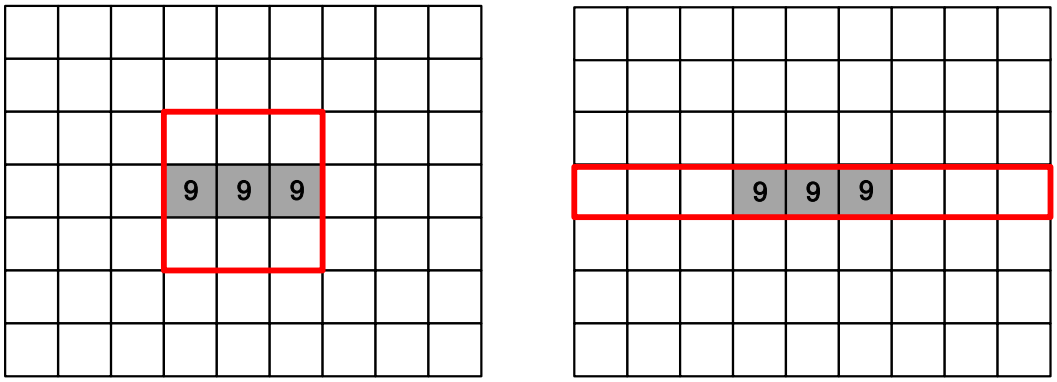
Require: Cell distribution of a window denoted by (l_x, l_y, u_x, u_y) is changed. The occupied area in bin $b_{i,j}$ is denoted by $o_{i,j}$. The grid size is $p \times q$. The occupied area in window $(0, 0, x, y)$ whose lower-left bin is $(0, 0)$ and upper-right bin is (x, y) is denoted by $O_{x,y}$.

Ensure: Look-up table is update.

if $l_x == 0 || l_y == 0$ **then**
  $O_{0,0} = o_{0,0}$;
  for $i = 1; i < p; ++i$ **do**
    $O_{i,0} = O_{i-1,0} + o_{i,0}$;
  end for
  for $j = 1; j < q; ++j$ **do**
    $O_{0,j} = O_{0,j-1} + o_{0,j}$;
  end for
end if
for $i = max\{l_x, 1\}; i <= u_x; ++i$ **do**
  for $j = max\{l_y, 1\}; j <= u_y; ++j$ **do**
    $O_{i,j} = O_{i-1,j} + O_{i,j-1} - O_{i-1,j-1} + o_{i,j}$;
  end for
end for
for $i = max\{l_x, 1\}; i <= u_x; ++i$ **do**
  for $j = u_y + 1; j < q; ++j$ **do**
    $O_{i,j} = O_{i-1,j} + O_{i,j-1} - O_{i-1.j-1} + o_{i,j}$
  end for
end for
for $i = max\{u_x, 1\} + 1; i < p; ++i$ **do**
  for $j = max\{l_y, 1\}; j <= u_y; ++j$ **do**
    $O_{i,j} = O_{i-1,j} + O_{i,j-1} - O_{i-1,j-1} + o_{i,j}$;
  end for
end for

2.3.5 Recursive bisection based cell spreading

Once the window for placement density hotspot is determined, the cells (within window) are evenly assigned to each bin (within window) in order to get a roughly legalized placement. During this cell-to-bin assignment, it is almost impossible to maintain the x- and y-directed relative cell positions simultaneously, due to the irregular distribution of placement density. To balance the loss of x and y-directed relative positions, the horizontal and vertical cut are applied alternatively similar to the slicing tree of [129]. When vertical cut is applied, the cells are sorted by their x positions, and the cells on left part are assigned to left child, the other cells are assigned to right child. While the horizontal cut is applied, the cells are sorted by their y positions, and cells on the bottom and top part are respectively assigned to left and right child. We define partitioning tree as follows.



(a) Reasonable aspect ratio.

(b) Unreasonable aspect ratio.

Figure 2.7 Aspect ratio of τ-enumerated window.

Definition 7. *For each window, its partitioning tree is a binary tree. Its node can be represented by four fields: a split line, a left child, a right child and an associated rectangle. Partition tree of a window satisfies the following conditions:*

- *The whole window is the associated rectangle of root.*
- *For each inner node, the split line should be one of the horizontal/vertical lines that divide the placement region into uniform bins.*
- *For each inner node, its associated rectangle is split into two parts by its split line. If its split line is horizontal, the above/below part is associated to its left/right child. Similarly, if its split line is vertical, the left/right part is associated to its left/right child.*
- *For each leaf node, its split line is invalid and its associated rectangle is one-to-one mapped to a bin.*

For each inner node, we try to balance the sizes of associated rectangles of its two children by choosing the split line which is closest to the middle horizontal or vertical line of its associated rectangle. A simple implementation of this cell-bin assignment is presented in Algorithm 4. At the beginning, only the root of partitioning tree is in the queue Q. In each iteration, a tree node is popped from Q and partitioned into two by a horizontal or vertical cut. During the

partitioning, the space utilization ratios of its two children's corresponding rectangles are closed to each other by properly allocating movable cells. The Algorithm 4 stops until the queue Q is empty, which means that the partitioning tree is constructed completely. For example, Fig. 2.8 is the partitioning tree for an 3×3 grid window.

Comparing with [129], there are several differences in our recursive bisection approach. Firstly, cut line should be strictly one of the split lines that divide placement region into uniform bins. Secondly, the leaf node should be strictly one-to-one mapped to a bin in order to control the placement density of each bin, which means that partitioning tree may not be a complete binary tree. Thirdly, the slice tree proposed by [129] is used to adjust cut line to handle with routing congestion, while our partitioning tree is served to achieve roughly legal placement where placement density of each bin is closed to design target.

Algorithm 4 Cell-to-bin assignment within window

Require: The set of cells S and window F (l_x, l_y, r_x, r_y).
Ensure: Each cell is assigned to exactly one bin.
1: respectively get the x and y-directed cell ordering;
2: determine the initial cut type T, it is vertical when $r_x - l_x > r_y - l_y$, otherwise horizontal;
3: create a root node R, push the quadruple (R, F, T, S) into a queue Q;
4: **while** Q is not empty **do**
5: pop (R, F, T, S) from Q;
6: partition the window and cell set, the results are $(F1, S1)$ and $(F2, S2)$, where $F = F1 \cup F2$ and $S = S1 \cup S2$;
7: create two children $C1, C2$ for inner node R;
8: change cut type T;
9: **if** $F1$ is not a bin and not empty **then** push ($C1$, $F1$, T, $S1$) into Q
10: **end if**
11: **if** $F2$ is not a bin and not empty **then** push ($C2$, $F2$, T, $S2$) into Q
12: **end if**
13: **end while**

2.3.5.1 Speedup technique

The runtime of Algorithm 4 is dominated by line 6 (partition inner tree node). For each inner tree node, sorting should be performed to partition cells. Therefore, the time complexity of Algorithm 4 is $O(nlog^2n)$, where n is the number of cells within window. Next, we will show a clever implementation of Algorithm 4. We can reduce the time complexity to $O((p+q)k + nlog(pq))$ for a $p \times q$ grid window, where k is a small constant.

There are two techniques to speedup Algorithm 4. Firstly, cells are not really moved during the construction of partitioning tree. For each cell, a path from the root to the leaf (the bin to which it is finally assigned) is maintained. So once the partitioning tree is constructed, each cell could be assigned to a bin by going through its path. For the convenience of computation, a bit sequence is maintained to denote this path. Secondly, the construction of partitioning tree is strictly level-by-level. To generate a new level of partitioning tree, the whole sorted cell list of window is just scanned once. Therefore, the total number of sorting is reduced to 2 [27].

Besides, we can take advantage of that the ranges of x and y coordinates of cells are known when the window is given. If we split every bin into k horizontal or vertical stripes like [27], the x and y-directed sorting can be done by bucket sort only losing little accuracy. And our experimental results show that the placement quality is not suffered if k is set to a reasonable value, which is 100 in POLAR.

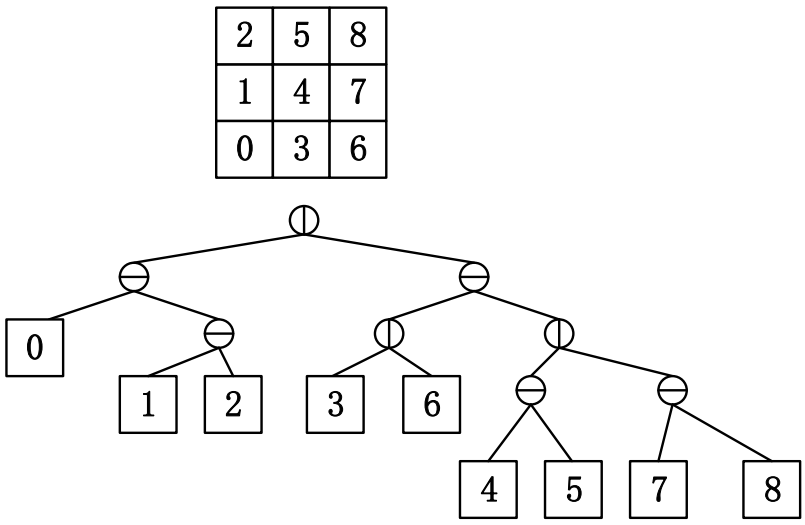


Figure 2.8 An example of partitioning tree for 3 × 3 window, the vertical cut line is first applied.

The detail of our speedup technique is presented in Algorithm 5. The inner nodes that have been partitioned are called **dead nodes**, otherwise **live nodes**. For each window, at the beginning, there is only one live node. The partitioning tree is constructed level by level, and a queue Q is used to maintain current live nodes. In each level, the same type of cut line is

applied to all the live nodes in lines 8-19. For any sub window F (l_x, l_y, u_x, u_y), it is divided into two subregions L (left subregion by vertical cut $\frac{l_x+u_x}{2}$, or bottom sub-region by horizontal cut $\frac{l_y+u_y}{2}$) and R (right subregion by vertical cut $\frac{l_x+u_x}{2}$, or top subregion by vertical cut $\frac{l_y+u_y}{2}$). The partitioning pivot is computed based on Formula 2.12.

$$\frac{A_L}{(A_L + A_R)} \times A_M \tag{2.12}$$

where A_L and A_R are the available area of L and R respectively, A_M is the total area of movable cells within F. There is a special case during the partitioning. If the sub window is a 1×2 (2×1) grid, the vertical (horizontal) cut line can not be applied apparently, since each leaf of partitioning tree is one-to-one mapped to a bin. Therefore, partitioning of this sub windows is delayed to the next level where the cut type is changed, we call this situation **level delay**.

An example is illustrated in Fig. 2.9. Assuming that the given window F_1 is a 3×3 grid, each bin has the same available space. The nine equal sized cells within F_1 are $\{C_1, C_2, \cdots, C_9\}$, the area of each cell is 1. The cells are sorted in ascending order according to their x- and y-coordinates respectively, the x-directed order is $(C_3, C_4, C_1, C_2, C_5, C_6, C_0, C_7, C_8)$, while the y-directed order is $(C_7, C_8, C_2, C_1, C_3, C_4, C_5, C_6, C_0)$.

In the first level, the vertical cut line is applied, according to Formula 2.12, the pivot is 3. The cell list is scanned in x-directed order, then C_3, C_4, C_1 should be assigned to left sub-window, others should be assigned to right sub-window, the partial path of each cell is updated. In the second level, the horizontal cut line is applied, the partitioning pivots for F_2 and F_3 are respectively 1 and 2. Then the cell list is scanned in y-directed order. The first one is C_7. Its partial path is "1", which means that it should be assigned to F_3 in the last level, and the current pivot of F_3 is 2, so it should be assigned to F_5 and its partial path is "10". The partial paths of other cells are updated similarly. In the third level, the vertical cut line is applied and the cell list is scanned in x-directed order. The first cell is C_3, its current partial path is "01". Since the associated rectangle of the node to which C_3 is belonged is F_4, and F_4 a 1×2 grid, a level delay happens and we update the partial path of C_3 to "010". The next cell is C_4. The level delay happens again and its partial path is updated accordingly. The next cell is C_1 and its partial path is "00". Since the associated rectangle of the node to which C_1 is belonged is

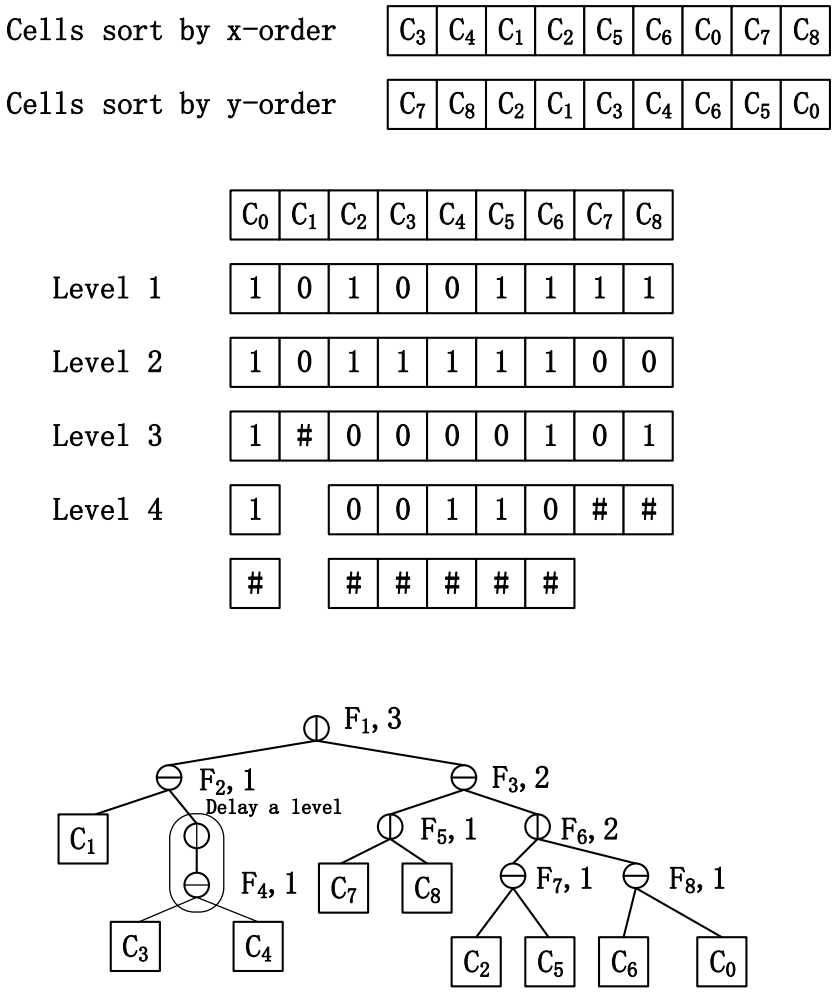


Figure 2.9 An example for recursive bisection based cell spreading with speedup technique.

already a bin, we do not update its partial path. The next cell is C_2 and its partial path is "11". In this case, we fetch the pivot of the node to which it is belonged and this pivot is 2. Therefore, C_2 should be assigned to left child and its partial path is updated to "110". The partial paths of other cells are updated similarly.

For any cell, once we know its path from root to leaf in partitioning tree, the bin which it is finally assigned to can be determined according to line 24-32 in Algorithm 5. For example, as shown in Fig. 2.9, the path of C_8 is "101" and cur_node in Algorithm 5 is set to root (represented by F_1) initially. The first bit of the path is "1", so cur_node is set to F_3 which is

Algorithm 5 Speed up technique for cell-to-bin assignment

Require: The set of cells S and window F (l_x, l_y, r_x, r_y)
Ensure: Each cell is assigned to exactly one bin.
1: perform x and y-directed cell sorting by bucket sort;
2: determine the initial cut type T, it is vertical when $r_x - l_x > r_y - l_y$, otherwise horizontal;
3: create a root node R, push the (R, F, *pivot*) into a queue Q;
4: **while** true **do**
5: $\Phi = \emptyset$; ▷ Φ stores the next level sub windows
6: **if** Q is empty **then** break; ▷ all the leaf nodes have been generated
7: **end if**
8: **while** Q is not empty **do**
9: pop (R, F, *pivot*) from Q;
10: **if** F is a 1×2 (2×1) grid && the cut is vertical (horizontal) **then**
11: $\Phi = \Phi \cup F$;
12: **else if** F is not bin **then**
13: partition F into $F1$ and $F2$ based on *pivot*;
14: create two children $C1$ and $C2$ for R;
15: calculate the pivot for $F1$ and $F2$, denoted by *pivot*1, *pivot*2;
16: $\Phi = \Phi \cup \{(C1, F1, pivot1), (C2, F2, pivot2)\}$;
17: update the bit sequence for each cell;
18: **end if**
19: **end while**
20: change the cut type T;
21: push each element of Φ into Q;
22: **end while**
23: **for** each cell in S **do**
24: cur_node := root of partitioning tree;
25: **for** each bit in the bit sequence of the cell **do**
26: **if** current bit is 0 **then**
27: cur_node: = cur_node's left child;
28: **else**
29: cur_node: = cur_node's right child;
30: **end if**
31: **end for**
32: assign this cell to the bin associated to cur_node;
33: **end for**

the right child of F_1. The second bit is "0", so cur_node is set to F_5 which is the left child of F_3. Finally, the last bit is "1", so it should be assigned to bin 6 in Fig. 2.8.

We analyze the time complexity of Algorithm 5 as follows. Line 1 needs $O((p+q)k+n)$. The level of partitioning tree is the same as its height, which is $O(log(pq))$. In each level, the whole cell list is scanned once by either x-directed or y-directed order, so the line 4-22 needs $O(nlog(pq))$. For each cell, $O(log(pq))$ is needed to scan its path (bit sequence). Therefore, the total time complexity of Algorithm 5 is $O((p+q)k + nlog(pq))$.

2.3.6 Cell spreading within bins

After recursive bisection based cell spreading, the information that which cell belongs to which bin is known. However, the concrete positions of cells in each bin are still undetermined. Here, similar with [27], we use a scaling method [103] to handle with this issue. To calculate x-coordinates of the cells within a bin, these cells are sorted in ascending order according to their original x-coordinates before look-ahead legalization. Then all these cells are put side by side in x-direction based on the above ordering. Since the total cell width may be longer than the width of bin, the x-coordinates of cells are scaled to make sure the centers of all the cells are between bin's left boundary and right boundary. Similarly, y-coordinates of the cells within a bin can be calculated. The details are presented in Algorithm 6.

Algorithm 6 Cell spreading in a bin

Require: The cells belong to the bin.
Ensure: The positions of cells.
1: Get the coordinate of bin's lower-left corner, denoted by (bin_x, bin_y) ;
2: Sort the cells in ascending order based on original x-coordinates before look-ahead legalization
3: $total\_width = 0$;
4: **for** each cell v **do**
5: $\quad total\_width += v$'s width;
6: **end for**
7: $cur\_pos_x = 0$;
8: **for** each cell v **do**
9: $\quad v$'s x-coordinate $= bin_x + \frac{cur\_pos_x}{total\_width}$;
10: $\quad cur\_pos_x += v$'s width;
11: **end for**
12: Calculate the y-coordinate of each cell by similar method;

Algorithm 7 Look-ahead legalization of POLAR

Require: A placement and target utilization θ.
Ensure: The density of each bin is closed to θ.
1: Build up the look-up table (see Formula 2.10).
2: Placement density hotspot detection (see Algorithm 1).
3: Sort all the density hotspots in descending order of density (the ratio of cell area and available area within hotspot).
4: **for** each density hotspot H **do**
5: $\quad$ In H, subtract the bins that have been covered by previous windows.
6: $\quad$ Find an expansion window F for H (see Algorithm 2).
7: $\quad$ Cell-to-bin assignment within F (see Algorithm 5).
8: $\quad$ Update the look-up table because the cell distribution of F is changed (see Algorithm 3).
9: **end for**
10: Spread the cells within bins (see Algorithm 6).

2.3.7 Complete algorithm of look-ahead legalization

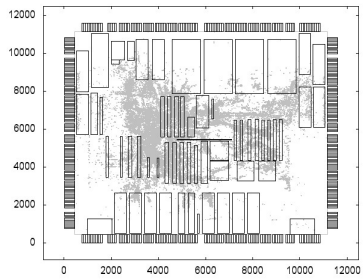
Algorithm 7 combines all the above together. The input could be any uneven placement. Firstly, the density estimation of given placement is performed to update the loop-up table mentioned in Section 2.3.4. Then Algorithm 1 is applied to find all the density hotspots. Density hotspots are sequentially handled according to their density. For each hotspot H, Algorithm 2 is used to find its window, within which Algorithm 5 finishes the cell-to-bin assignment. Since some bins of hotspot H may have been already covered by previous windows and are not overflow any more, they are subtracted from H. Note that the cell distribution of placement is changed after applying Algorithm 5, so the look-up table should be updated by Algorithm 3. At the end, cells within each bin are simply spread out by Algorithm 6. To illustrate our look-ahead legalization, placement migration of circuit adaptec1 is shown in Fig. 2.10, where both lower bound and upper bound placement are presented.

2.3.8 Force modulation

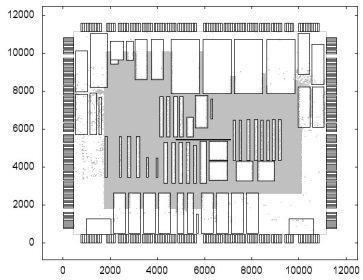
After all the anchors of cells are determined, the linear system 2.3 is updated and solved again. Firstly, the B2B net model is refreshed based on anchor positions of cells. Then, for each movable cell, a two-pin pseudo net connecting it to its anchor is added into the spring system. The weight ρ of pseudo net is calculated according to Formula 2.13, where i means the ith iteration of density driven placement.

$$\rho = \begin{cases} \varepsilon & \text{if } i = 0 \\ \varepsilon * \alpha^{i-1} & \text{if } 1 \leq i \leq 20 \\ \varepsilon * \alpha^{19} * \beta^{i-20} & \text{if } i > 20 \end{cases} \tag{2.13}$$

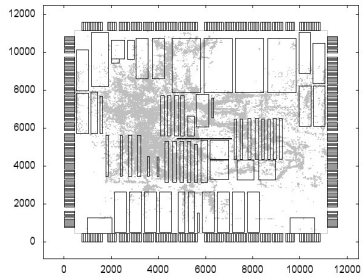
It is a two-stage force modulation, where ε is a small value and $\beta > \alpha$. At the early stage, ρ is small in order to avoid significant change of placement. While at the latter stage, ρ is increased more quickly to speedup the convergence. For ISPD 2005 and 2006 benchmark suites [124, 125], the default value of $\varepsilon, \alpha, \beta$ are respectively 1e-5, 1.05, 1.15.



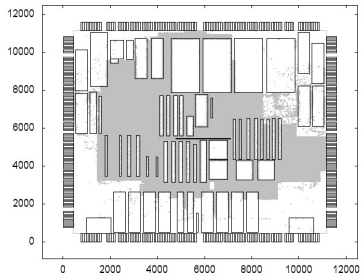
(a) The lower bound placement in the 5th iteration.



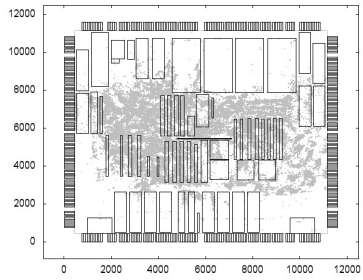
(b) The upper bound placement in the 5th iteration.



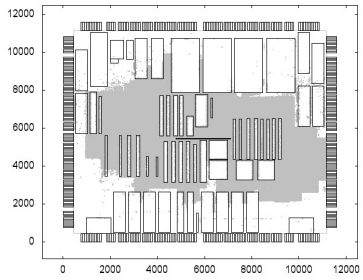
(c) The lower bound placement in the 15th iteration.



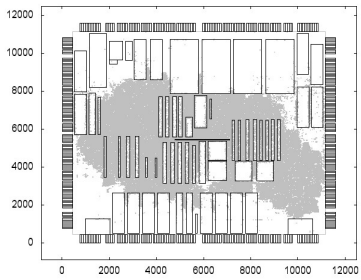
(d) The upper bound placement in the 15th iteration.



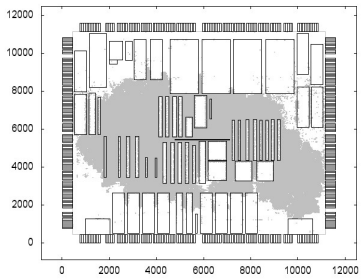
(e) The lower bound placement in the 30th iteration.



(f) The upper bound placement in the 30th iteration.



(g) The lower bound placement in the last iteration.



(h) The upper bound placement in the last iteration.

Figure 2.10 Placement migration of circuit adaptec1.

2.3.9 Handling with movable macros

To handle with movable macros, there are many existent methods such as [130, 35, 36, 37]. We used a light-weight approach which can be easily integrated into Algorithm 7 as shown in Algorithm 8.

The idea originates from [130] and [28]. Each movable macro is sliced into a set of equal-sized pieces. Similar with [28], these sliced pieces share the same central position as the movable macro which they are from just before look-ahead legalization. Different from [130], these sliced pieces are not connected by fake nets, so this approach does not touch matrix generation. In other words, exactly the same as standard cell, each movable macro is treated as an entity rather than a set of sliced pieces during matrix generation. The size of each piece is about the average size of standard cells. Since POLAR tries to maintain the relative positions of cells, most of the sliced pieces belonged to the same macro would be very close to each other. And the position of movable macro is the gravity center of its sliced pieces just after look-ahead legalization. Therefore, even few of its sliced pieces maybe a little far away from the majority of others, it has little impact on the final position of movable macro.

Algorithm 8 Handle with movable macros

1: Shredding moveable macros into slices;
2: Apply Algorithm 7;
3: Resembling the slices into macros;

2.4 Experimental Results

POLAR is implemented in C++. The binaries of several modern academic placers such as NTUPlacer3[26], mPL6 [24], FastPlace3 [22], SimPL [27] and ComPLx [28] are also obtained. However, MAPLE [117] is not available since it is a commercial tool. All results (excepts those of MAPLE) are generated in the same platform, which is a Linux PC with 16GB of memory and Intel Core-i3 3.3GHz CPU. Note that some academic placers such as SimPL and ComPLx have paralleled implementation, while some other do not. To make the comparison fair, we only allow to use one core for all placers. The ISPD 2005 benchmark suite [124] and ISPD 2006 benchmark suite [125] are used to verify the efficiency of POLAR.

2.4.1 Benchmark characteristics

The characteristics of ISPD 2005 benchmark suite and ISPD 2006 benchmark suite are listed in Table 2.1 and Table 2.2. For the ISPD 2005 benchmark suite, the density target θ is set to 1, which means that the whitespace in each bin could be occupied by movable cells completely. However, for the ISPD 2006 benchmark suite, the density target of each benchmark is fixed by contest organizers. The range of number of cells (including both movable cells and fixed macros/pins) in those benchmarks is from 0.2 millions to 2.5 millions.

Table 2.1 Characteristics of ISPD 2005 benchmark suite

benchmark	# of V	# of E	design utility	density target
adaptec1	211447	221142	57.34%	100%
adaptec2	255023	266009	44.32%	100%
adaptec3	451650	466758	33.52%	100%
adaptec4	496045	515951	27.14%	100%
bigblue1	278164	284479	44.67%	100%
bigblue2	557866	577235	37.78%	100%
bigblue3	1096812	1123170	56.48%	100%
bigblue4	2177353	2229886	42.29%	100%

Table 2.2 Characteristics of ISPD 2006 benchmark suite

benchmark	# of V	# of E	design utility	density target
adaptec5	843128	867798	49.97%	50%
newblue1	330474	338901	83.20%	80%
newblue2	441516	465219	61.66%	90%
newblue3	494011	552199	26.30%	80%
newblue4	646139	637051	46.45%	50%
newblue5	1233058	1284251	49.55%	50%
newblue6	1255039	1288443	38.78%	80%
newblue7	2507954	2636820	49.31%	80%

2.4.2 Comparison on ISPD2005 benchmark suite

For the ISPD 2005 benchmark suite, the quality of placement is measured by wirelength. The experimental results are presented in Table 2.3. On average, POLAR achieves the best quality (respectively improve the wirelength by 7.87%, 2.68%, 6.67%, 2.32%, 1.30% and 0.14%

compared with NTUPlacer3, mPL6, FastPlace3, SimPL, ComPLx and MAPLE) and it is also very fast. Comparing with SimPL which is fastest one, POLAR improves the wirelengh by 2.32% at the cost of 6% increase in the runtime. Comparing with MAPLE which produces similar wirelength with POLAR, we get 6.73× speedup [2].

2.4.3 Comparison on ISPD2006 benchmark suite

In the ISPD 2006 benchmark suite, the circuit newblue1 has several movable macros. The quality of placement is measured by scaled wirelength. The scaled wirelength is composed of two parts: wirelength and the penalty of overflow. The contest official script [125] is used to calculate the scaled wirelength, and the experimental results are presented in Table 2.4 and Table 2.5. On average, POLAR achieves improvement of 3.54%, 5.74%, 11.68%, 0.79% and 0.44% on scaled wirelength versus NTUPlace3, mPL6, FastPlace3, ComPLx and MAPLE. We do not have the results of SimPL, since it currently does not support to run on ISPD 2006 benchmark suite. For the runtime, on average, POLAR is 2.59×, 8.91×, 1.00×, 1.05× faster than NTUPlacer3, mPL6, FastPlace3 and ComPLx. The runtime of MAPLE on ISPD 2006 benchmark suite was not reported in [117], so we cannot compare POLAR with it either directly or indirectly.

2.4.4 Runtime analysis

The runtime breakdown of POLAR is shown in Table 2.6. It is divided into three components: global placement, legalization, and detailed placement. The runtime of global placement is further divided into three parts: PCG, look-ahead legalization (which includes hotspot detection, window enumeration and recursive bisection based cell spreading) and others (e.g. B2B net model update, wirelength calculation and I/O).

On average, global placement takes about 76% of total runtime, while legalization and detailed placement respectively take about 7% and 17% of total runtime. In global placement stage, PCG takes about 50% of total runtime, look-ahead legalization takes 14% (window

[2]The binary of MAPLE is not released, so MAPLE's runtime is scaled according to [117] which show that it is 7.14 × slower than SimPL

Table 2.3 Placement quality comparison on ISPD 2005 benchmark suite

benchmark	NTUPlace3		mPL6		FastPlace		SimPL		ComPLx		MAPLE		POLAR	
	HPWL	runtime	HPWL	runtime	HPWL	runtime	HPWL	runtime	HPWL	runtime	HPWL	runtime	HPWL	runtime
adaptec1	79.83	5.11	77.29	19.3	78.66	2.61	77.53	2.43	77.79	2.64	76.36	19.3	76.54	2.43
adaptec2	90.08	6.73	90.02	20.1	94.06	3.57	91.11	3.50	88.97	3.27	88.97	24.2	86.57	3.37
adaptec3	232.72	13.5	207.0	62.7	214.13	8.10	203.79	6.75	203.57	7.58	209.78	50.7	202.8	7.17
adaptec4	215	16.4	188.87	59.2	197.5	7.43	184.75	5.18	183.22	6.65	179.91	49.2	182.93	7.13
bigblue1	96.94	9.72	96.18	24.4	96.67	3.73	95.59	4.46	95.3	5.20	93.74	24.7	94.36	3.40
bigblue2	158.26	24.2	148.91	68.1	155.74	6.50	145.87	5.61	145.39	7.03	144.45	43.8	143.89	8.13
bigblue3	343.57	27.8	335.53	93.2	365.16	18.8	351.65	17.7	337.96	18.1	323.05	89.5	323.05	16.5
bigblue4	825.48	81.0	814.13	212	836.2	34.7	791.29	26.9	788.8	35.8	775.71	231	791.27	34.9
Norm.	+7.87%	2.26×	+2.68%	7.28×	+6.67%	1.04×	+2.32%	0.94×	+1.30%	1.07×	+0.14%	6.73×	+0.00%	1.00×

enumeration uses 2% and recursive bisection based cell spreading uses 9%) and others take 12% of total runtime.

Table 2.4 Placement quality comparison on ISPD 2006 benchmark suite

benchmark	NTUPlace3		mPL6		FastPlace3		ComPLx		MAPLE		POLAR	
	sWL	overflow	sWL	overflow	sWL	overflow	sWL	overflow	sWL	overflow	sWL	overflow
adaptec5	444.41	28.51	428.31	1.03	472.72	8.17	415.77	1.93	407.33	4.76	411.91	6.42
newblue1	61.01	0.70	72.62	9.02	74.11	1.04	64.75	1.02	69.25	1.05	67.2	1.11
newblue2	194.8	1.82	201.91	1.44	206.04	1.00	193.06	1.05	191.66	1.01	192.8	1.18
newblue3	275.08	0.05	285.26	0.66	297.46	0.55	274.64	0.93	268.07	0.77	270.58	1.01
newblue4	296.62	13.6	298.2	1.70	308.35	4.22	292.82	1.45	282.49	5.86	282.67	3.31
newblue5	537.92	20.3	535.8	1.47	621.47	7.21	507.74	1.76	515.04	4.05	502.96	5.36
newblue6	534.96	0.28	523.47	1.41	549.87	1.02	501.05	1.14	494.82	1.08	497.86	1.39
newblue7	1096.16	2.01	1085.68	1.19	1105.43	1.30	1041.21	1.40	1032.6	1.70	1025.4	1.01
Norm.	+3.54%	2.07	+5.74%	1.62	+11.68%	1.01	+0.79%	0.73	+0.44%	1.01	+0.00%	1.00

Table 2.5 Runtime comparison on ISPD 2006 benchmark suite

benchmark	NTUPlace3	mPL6	FastPlace3	ComPLx	POLAR
adaptec5	45.2	118.2	16.6	15.0	13.7
newblue1	8.9	27.1	4.2	3.6	6.9
newblue2	17.4	66.4	6.7	8.6	6.9
newblue3	15.1	102.4	8.6	7.7	7.1
newblue4	26.4	89.1	9.2	9.6	9.0
newblue5	58.1	161.8	21.4	23.6	21.5
newblue6	50.0	133.9	21.1	19.1	17.8
newblue7	120.3	377.1	33.2	47.4	38.5
Norm.	2.59×	8.91×	1.00×	1.05×	1.00×

2.5 Conclusions

In this chapter we have proposed a high performance mixed-size wirelength-driven placer called POLAR. It adopts the popular framework of legalization. An elegant and effective algorithm for look-ahead legalization is proposed. The experimental results on ISPD 2005 benchmark suite and ISPD 2006 benchmark suite verify that our placer is very comparable to the state-of-the-art placers in both runtime and placement quality.

Table 2.6 Runtime breakdown of POLAR

benchmark	global placement					legalization	DP	total runtime
	PCG	LAL			others			
		window enumeration	cell spreading	others				
adaptec1	66	3	11	5	22	7	32	146
adaptec2	86	3	15	7	25	7	59	202
adaptec3	208	14	29	9	42	21	107	430
adaptec4	203	8	35	11	41	19	111	428
bigblue1	95	3	16	6	27	6	51	204
bigblue2	279	9	30	13	49	32	76	488
bigblue3	436	12	102	29	87	61	261	988
bigblue4	1072	46	187	50	215	122	400	2092
adaptec5	431	15	91	26	90	103	63	819
newblue1	260	1	52	7	55	23	19	417
newblue2	196	5	42	10	43	89	27	412
newblue3	175	29	31	28	40	26	98	427
newblue4	273	7	55	18	58	48	82	541
newblue5	630	20	173	27	127	94	217	1288
newblue6	610	15	120	41	127	64	92	1069
newblue7	1253	129	242	80	245	232	128	2309
Normalize	0.50	0.02	0.09	0.03	0.12	0.07	0.17	1.00

CHAPTER 3. POLAR 2.0: AN EFFICIENT ROUTABILITY-DRIVEN PLACER

A wirelength-driven placer without considering routability would lead to unroutable results. To mitigate routing congestion, there are two basic approaches: (i) minimizing the routing demand; (ii) distributing the routing demand properly. In this chapter, we propose a new placer POLAR 2.0 [45] emphasizing both approaches. To minimize the routing demand, POLAR 2.0 attaches very high importance to maintaining a good wirelength-driven placement in the global placement stage. To distribute the routing demand, cells in congested regions are spread out by a novel routability-driven rough legalization in a global manner and by a history based cell inflation technique in a local manner. The experimental results based on ICCAD 2012 contest benchmark suite [51] show that POLAR 2.0 outperforms all published academic routability-driven placers.

3.1 Introduction

Placement is one of the most important and ancient problems in Electronic Design Automation (EDA). Its quality has been greatly improved during the last two decades. However, with the gradually increasing scale of design, a high quality while extremely fast placer is still in urgent need. Besides, [131, 132] pointed out that the commonly used wirelength metric might not capture the key aspects of solution quality. Overemphasis of wirelength as in traditional placement formulation inevitably results in bad quality in other metrics such as power, timing and routability, although optimizing wirelength is beneficial to those metrics to some extent.

Among varieties of metrics, routability is now becoming more and more important due to a significant mismatch between the objectives of wirelength and routing congestion. A

wirelength-driven placer without considering routability usually leads to irresolvable routing congestion problem. In the recent years, a series of contests (ISPD 2011, DAC 2012, ICCAD 2012) were held to promote the research in routability-driven placement, and some academic routability-driven placers [44, 133, 42, 24, 43, 134, 46] such as coPR [133], Ripple 2.0 [43] and NTUplace4h [46] were produced. Besides, SRP [41] and Ropt [135] try to refine routability-driven placement by using the routing information feedbacked by global router.

There are two challenges in routability-driven placement problem. The first challenge is that the routing congestion is expected to be detected accurately in short runtime. Since directly invoking the whole routing process in the global placement stage is very time consuming, many routing congestion estimation methods were proposed. Fast global routers such as FastRoute [136] and BFG-R [137] have been incorporated into some placers [44, 133, 42, 43] to achieve relatively accurate estimation. RUDY [138] adopts a L-shaped probability model and half perimeter wirelength (HPWL) to estimate the actual routing demand. Ripple [42] and NTUPlace4h [46] further extend RUDY's method. Ripple utilizes rectilinear minimum spanning tree (RSMT) to replace HWPL, while NTUPlace4h applies Guassian smoothing to smooth the L-shaped approximation model. The second challenge is that the cells within routing congestion region should be spread out to balance the routing supply and routing demand. Many placers [47, 139, 49, 42, 43, 44, 133, 24] apply cell inflation. CROP [140] adjusts the boundary of each G-Cell to make sure it has enough available area and routing supply. NTUPlace4h formulates routing congestion as an additional constraint into its non-linear programming framework.

In this chapter, we propose a new routability-driven placer, POLAR 2.0, which mitigates routing congestion by the following two basic approaches: (i) minimizing the routing demand; (ii) spreading the routing demand properly. To minimize the routing demand, the new placer attaches very high importance to maintaining a good wirelength-driven placement in the global placement stage. To distribute the routing demand, cells in congested regions are spread out by a novel routability-driven rough legalization in a global manner and by a history based cell inflation technique in a local manner. Experimental results on ICCAD 2012 Contest benchmark suite show that POLAR 2.0 outperforms all published academic routability-driven placers.

Compared with SimPLR [44], CoPR [133], Ripple 2.0 [43] and NTUPlace4h [46], our placer respectively achieves 5.1%, 3.0%, 2.8% and 1.1% improvement on the scaled HPWL.

The key ideas of POLAR 2.0 are highlighted as follows.

- Maintaining a good wirelengh-driven placement is attached very high importance in our routability-driven optimization flow, in order to minimize the routing demand.
- A novel routability-driven rough legalization is applied to distribute the routing demand. In this technique, routing congestion on the horizontal and vertical directions are handled separately, and the routing congestion regions are effectively spread out in a global manner.
- A history based cell inflation is adopted as a complement in a local manner. Different from some previous cell inflation techniques, the inflation amount is accumulated and kept until the global placement is finished.

The remainder of this chapter is organized as follows. Section 3.2 reviews the preliminaries. Section 3.3 presents the framework of POLAR 2.0. Section 3.4 illustrates POLAR 2.0's algorithm. Section 3.5 shows the experimental results. Finally, Section 3.6 are the conclusions.

3.2 Preliminaries

The routability-driven placement relies on the traditional wirelength-driven placement engine. A circuit can be represented by a hypergraph $G = (V, E)$, where V is the set of cells and E is the set of nets. The placement tries to determine the physical positions of the cells without violating the placement density constraints. We denote the x-coordinates of cells by a vector $\mathbf{x} = \left(x_1, x_2, \cdots, x_{|V|}\right)$, and y-coordinates by $\mathbf{y} = \left(y_1, y_2, \cdots, y_{|V|}\right)$, the objective is to minimize the half-perimeter wirelength (HPWL):

$$\text{HPWL}(\mathbf{x}, \mathbf{y}) = \Sigma_{e \in E}[\max_{i \in e} x_i - \min_{i \in e} x_i + \max_{i \in e} y_i - \min_{i \in e} y_i] \tag{3.1}$$

POLAR 2.0 is a natural extension of POLAR [33]. POLAR adopts the rough legalization idea [27], but rough legalization is realized in a different manner: for each density hotspot,

POLAR enumerates the optional reasonable windows to find the smallest one that can accommodate all the cells within it, and then a tree-based bisection method spreads the cells evenly.

A good wirelength-driven placement is beneficial to router, since the wirelength has direct influence on the routing demand. Usually, better total wirelength means less total routing demand. However, excessively optimizing the wirelength would lead to routing congestion, since the cells which have lots of connections are pulled together resulting into that the local routing demand substantially exceeds the local routing supply.

Routability-driven placement essentially is to distribute the routing demand rationally according to the routing supply. To simplify the routability formulation, the routing resources are given by a 2-D $m \times n$ mesh, since 3-D mesh can be easily transformed to 2-D mesh by accumulating the routing resources of different layers. The grid in the 2-D mesh is usually called G-Cell, which is denoted by a coordinate (x, y), where $0 \leq x < m$ and $0 \leq y < n$. For any pair of adjacent horizontal G-Cells, the connecting routing channel is called H-edge, while for any pair of adjacent vertical G-Cells, the connecting routing channel is called V-edge. Therefore, for each G-Cell, it is at most associated to two H/V-edges respectively. And the number of trunks in each H/V-edge is fixed as the routing supply. As shown in Fig. 3.1, it is a 4×4 2-D mesh, for the G-Cell $(1, 2)$, the two associated H-edges are colored red, while the two associated V-edges are colored blue. The horizontal/vertical routing supply(H/V-supply) of G-Cell is defined as the total number of trunks in its associated H/V-edges. The horizontal/vertical routing demand(H/V-demand) of G-Cell is defined as the number of wires that goes through its associated H/V-edges, the H-demand and V-demand of G-Cell $(2, 2)$ are respectively 2 and 2.

To capture a clear picture of design routability, [50] introduced average congestion of G-Cell edges(ACE). Optimizing ACE not only minimizes the total routing overflow but also produces rational distribution of the routing demand. In this chapter, the same as previous works [44, 133, 43, 46], the objective of routability-driven placement is to minimize both HPWL and ACE.

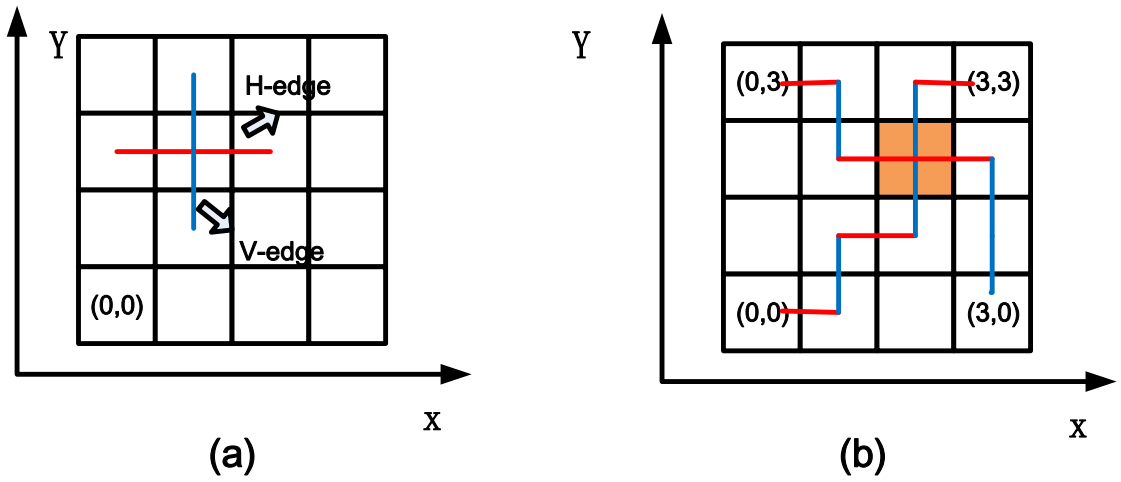


Figure 3.1 The 2-D routing mesh.

3.3 Overview

There are two basic approaches to optimize the routability: (i) minimizing the routing demand; (ii) spreading the routing demand properly. On one hand, since routing overflow is calculated by routing demand minus routing supply, roughly speaking, minimizing routing demand is beneficial to decrease routing overflow. On the other hand, for each H/V-edge, its routing demand is expected to be not higher than its routing supply. Therefore, the distribution of routing demand should be done properly based on the known distribution of routing supply.

POLAR 2.0 targets on the above two approaches to optimize the routability. Its overview is presented in Fig. 3.2. The whole placement is partitioned into three stages: (i) wirelength-driven seed placement generation; (ii) routability-driven cell spreading; (iii) post-global placement.

To minimize the routing demand, in POLAR 2.0, maintaining a good wirelength-driven placement is attached very high importance. In the first stage, POLAR [33] is used to generate a good wirelength-driven seed placement: the global placement loop of POLAR is not stopped until the number of iterations is greater than 50 and the gap between the upper bound wirelengh

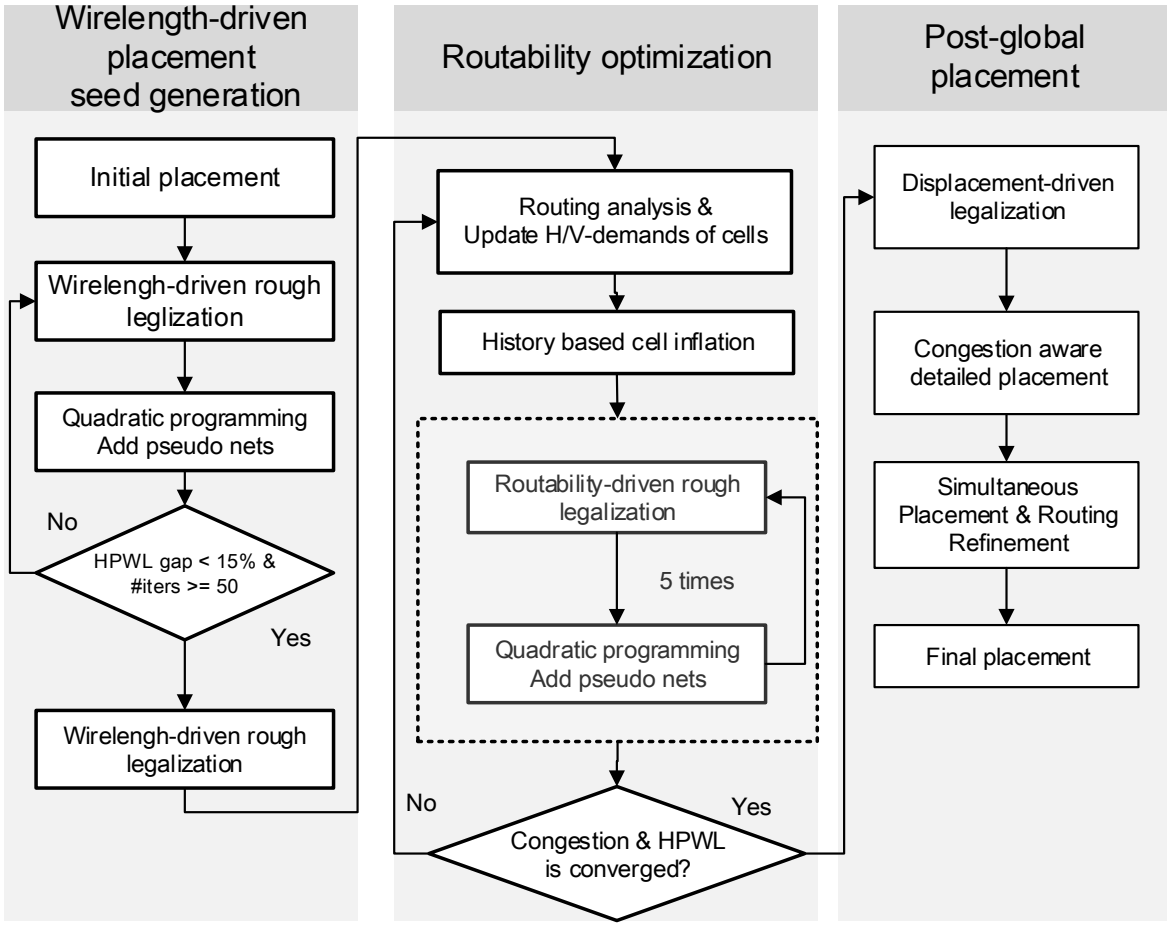


Figure 3.2 The overview of POLAR 2.0.

and the lower bound wirelength is less than 15%. Besides, in routability-driven cell spreading stage, for each round of routing analysis, the steps of quadratic programming and routability-driven rough legalization are iterated 5 times to maintain a good wirelength-driven placement.

In the routability-driven cell spreading stage, routing analysis is applied to calculate H/V-demands of G-Cells, and to model the migration of routing demand, H/V-demands of G-Cells are amortized to H/V-demands of movable cells. Based on movable cells' H/V-demands, POLAR 2.0 simultaneously distributes area demand, horizontal routing demand and vertical routing demand by the following two approaches. Firstly, in a global manner, we propose a routability-driven rough legalization which is a natural extension of POLAR's [33] rough legalization idea. During routability-driven rough legalization, both area and routing congestion

hotspots are detected. For each hotspot, the smallest window (expansion region) which has enough area and routing resources to satisfy all demands of the enclosed cells is searched by enumeration. Then a tree based bisection spreading technique is applied to distribute those cells within the window. Secondly, to avoid local routing congestion when distributing the cells within the window, a history based cell inflation technique is proposed. The details of this stage are presented in Section 3.4.

Finally, in the post-global placement stage, we adopt the same method as Ripple 2.0's [43], which has three components: (i) displacement-driven legalization, (ii) congestion aware detailed placement and (iii) simultaneous placement and routing refinement [41].

3.4 Routability Optimization

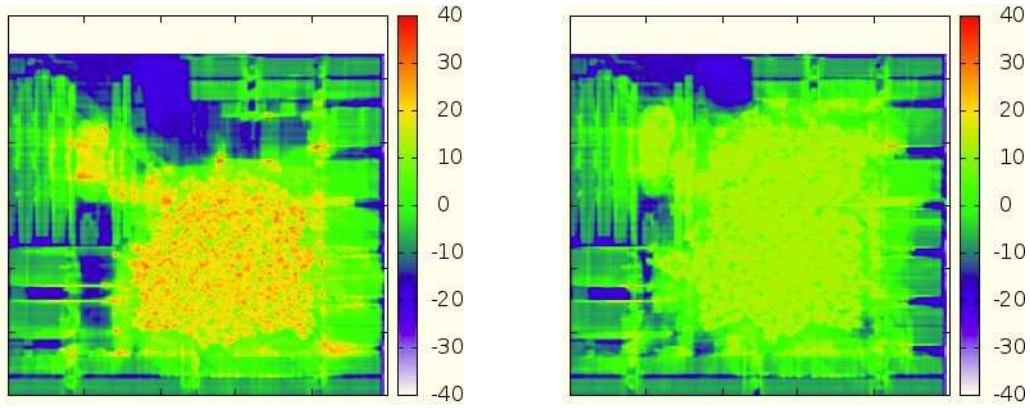
3.4.1 Routing analysis

3.4.1.1 Routing supply calculation

3-D routing can easily be transformed to 2-D routing by accumulating the total routing resources of all metal layers. Since different metal layers have different wire pitches, we need to sum up the number of tracks of each metal layer. In addition, there are many fixed routing blockages occupying the routing resources on the metal layers, the supply of H/V-edges need to exclude these blocked routing resources. The work [43] gives the details about the transformation from 3-D routing to 2-D routing.

3.4.1.2 Routing demand estimation

Different from the routing supply, the routing demand relays on the placement and routing solution. To calculate the exact routing demand, legalized placement and detailed routing are necessary. However, invoking legalization and detailed router during the global placement stage is very time consuming. Therefore, in POLAR 2.0, the routing demand is estimated based on roughly legalized placement and global routing instead: we use roughly legalized placement to calculate the pin locations, and then the congestion aware pattern routing of FastRoute [136] is applied to estimate the routing demand.



(a) (b)

Figure 3.3 The congestion map with/without using routability-driven rough legalization.

When the cells are moved, routing demand would be migrated. During the routability optimization, POLAR 2.0 maintains a good wirelength-driven placement, in which the relative positions of cells are only allowed to modified a little bit. Under this condition, for most of the cells, moving around would not change its H/V-demand much.

To trade off accuracy and runtime, we perform routing demand estimation infrequently. As shown in Fig. 3.2, the routing demands are updated only once every five times the placement solution is refined. To model the migration of routing demands, we associate the demands to movable cells by introducing two new attributes, the horizontal and vertical routing demand (H/V-demand), for each movable cell. Consider a movable cell i located in G-Cell j. Let the H/V-demand of G-Cell j be denoted by HD_j and VD_j, respectively, and the number of movable cells within G-Cell j be denoted by k_j. Then the H/V-demand of cell i, denoted respectively by hd_i and vd_i, is given by Formula 3.2 and 3.3:

$$hd_i = \frac{HD_j}{k_j} \tag{3.2}$$

$$vd_i = \frac{VD_j}{k_j} \tag{3.3}$$

3.4.2 Routability-driven rough legalization

After routing analysis, each movable cell has three attributes: area demand, H-demand and V-demand. And the routability-driven placement essentially is to distribute these three demands based on the given supplies (available area, horizontal routing supplies, vertical routing supplies). To realize this goal in a global manner, we propose a novel routability-driven rough legalization technique.

Algorithm 9 Routability-driven rough legalization

Require: The H/V-demands of movable cells are known, placement is rough legalized.
Ensure: Good wirelengh is maintained, routability is optimized.
1: Detect the area/routing congestion hotspots; ▷ The method is similar to POLAR's [33]
2: **for** each hotspot s **do**
3: $\Gamma = \emptyset$;
4: **for** each window w whose geometrical center is the same as s **do**
5: as = the available area of w;
6: hs = total H-supplies of the G-Cells contained by w;
7: vs = total V-supplies of the G-Cells contained by w;
8: ad = total areas of the movable cells located in w;
9: hd = total H-demands of the movable cells located in w;
10: vd = total V-demands of the movable cells located in w;
11: rt = the aspect ratio of w;
12: **if** $as \geq ad$ && $hs \geq \alpha \times hd$ && $vs \geq \alpha \times vd$ && $rt \in \left[\frac{1}{3}, 3\right]$ **then**
13: $\Gamma = \Gamma \cup \{w\}$;
14: **end if**
15: **end for**
16: Distribute the cells within the minimal window of Γ by tree based bisection [33].
17: **end for**

The pseudo code of routability-driven rough legalization is presented in Algorithm 9. In the original POLAR's [33] rough legalization, for each placement density hotspot, a minimal expansion window is found by enumerating the ones which have enough available area and reasonable aspect ratio, and then the cells are spread out evenly by a tree-based bisection within the chosen window. While, in the routability-driven rough legalization, not only placement density hotspots, but also routing congestion hotspots are detected. Besides, any expansion window w should have enough Horizontal and vertical routing supplies by satisfying the following two additional constraints.

$$\Sigma_{j \in w} HS_j \geq \alpha \times \Sigma_{i \in w} hd_i \tag{3.4}$$

$$\Sigma_{j \in w} VS_j \geq \alpha \times \Sigma_{i \in w} hd_i \tag{3.5}$$

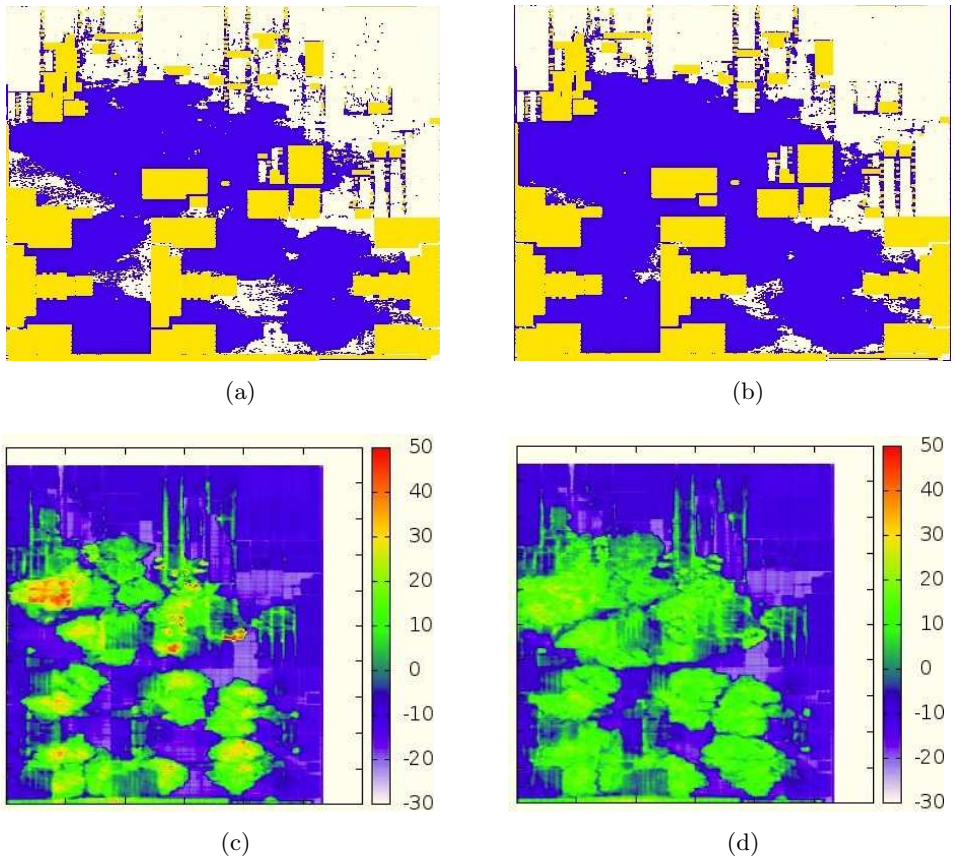
Where HS_j and VS_j are respectively the routing supplies of the associated H-edges and V-edges of G-Cell j, α is a parameter due to the inaccuracy of routing estimation. When α is higher than 1, it means that the routing congestion is underestimated; on the contrary, α is less than 1 means that the routing congestion is overestimated. Experimental results verify that α is less than 1, mainly because the time consuming maze routing is not used during routing analysis in POLAR 2.0.

This routability-driven rough legalization technique can effectively distribute the routing demand by mitigating the routing demand to the places which have enough routing supply without being used. Fig. 3.3 shows the congestion map of benchmark superblue16 with/without this technique. Fig. 3.3(a) is the one with POLAR's [33] rough legalization instead of POLAR 2.0's routability-driven rough legalization during routability-driven cell spreading stage. Note that the congestion value is scaled from -40 to 40, higher value means more congestion. The congestion map was drew based on the results of FastRoute's pattern routing [136]. It can be seen that with this technique, the routing congestion is spread out so that the scaled HPWL is significantly improved.

3.4.3 History based cell inflation

For any design, the distribution of area supply/demand, horizontal routing supply/demand and vertical routing supply/demand are usually not the same. The tree based bisection spreading technique used in routability-driven rough legalization only distributes cells evenly according to area supply/demand. Therefore, for some G-Cells, there are enough available areas to accommodate its enclosed cells, but may no enough horizontal/vertical routing resources to satisfy the routing demands of its enclosed cells. To avoid local routing congestion that routability-driven rough legalization cannot resolve, similear to previous works, the routing demands of some cells are transformed into inflated area by a history based cell inflation.

The pseudo code of history based cell inflation is presented in Algorithm 10. The principle is that only the movable cells located in the most congested G-Cells are inflated by a small



(a) (b)

(c) (d)

Figure 3.4 Density map shows effectiveness of POLAR 2.0.

Algorithm 10 History based cell inflation

Require: Routing analysis is just done intermediately.
Ensure: The movable cells located in the most congested G-Cells are inflated by a small ratio.

$\phi = \emptyset$;
for any G-Cell j **do**
    HD_j = the H-demand of G-Cell j;
    VD_j = the V-demand of G-Cell j;
    HS_j = the H-supply of G-Cell j;
    VS_j = the V-supply of G-Cell j;
    if $HD_j > HS_j$ **then**
        add the ordered pair $(j, HD_j - HS_j)$ into ϕ;
    end if
    if $VD_j > VS_j$ **then**
        add the ordered pair $(j, VD_j - VS_j)$ into ϕ;
    end if
end for
Sort the ϕ based on the overflow (the second item of ordered pair) in descending order;
for each ordered pair t in the top 10% of sorted ϕ **do**
    for each movable cell i in G-Cell $t.first$ **do**
        Inflate the area of cell i by 10%;
    end for
end for

ratio and the inflation is accumulated until the routability cannot be improved. Its insight is derived from the history based global routing technique [141] (In global routing, detour is not preferred, but usually inevitable. To route a design, [141] adds a big penalty to detour at the beginning to see whether all the nets can be routed without overflow. If the answer is no, then the detour penalty is decreased slightly and rerouteing is performed. This process is continued until all the nets are finally routed without overflow.) This approach is similar to other cell inflation techniques [47, 139, 49, 42, 43, 44, 133, 24] functionally. It is simple and works well according to our experimental results.

3.5 Experimental Results

POLAR 2.0 was implemented in C++ and complied by g++-4.7.2. The benchmarks of ICCAD 2012 contest [51] are ran on a Linux PC with Intel Xeon X5550 2.67GHz CPU and 16GB RAM to verify the efficiency of POLAR 2.0. Routability evaluation is performed by official script in the ICCAD 2012 contest [51]. The placement solution is routed by the designate global router-NCTRgr [142]. The scaled wirelength is calculated according to HPWL and ACE [50] penalty.

Table 3.1 Runtime breakdown on ICCAD 2012 benchmarks

Benchmark	Wirelengh-driven seed generation	Routability optimization			Post-global placement	Total runtime (s)	time ratio POLAR 2.0/POLAR[33]
		Routing analysis	cell spreading	others			
superblue1	411	84	564	273	280	1612	2.49
superblue3	425	75	534	295	431	1760	2.33
superblue4	240	58	490	267	205	1260	2.36
superblue5	328	55	162	141	414	1100	1.69
superblue7	814	61	375	252	361	1863	1.66
superblue10	579	119	316	243	1689	2946	3.09
superblue16	334	41	240	121	292	1028	1.83
superblue18	266	116	420	248	228	1278	2.59
Normalize	0.27	0.06	0.26	0.15	0.26	1.00	2.26

3.5.1 Runtime analysis

The runtime of our placer is shown in Table 3.1. It is broken down into three components: wirelengh-driven placement seed generation, routability optimization which includes routing analysis, history based cell inflation and window based cell spreading, and post-global placement. On average, routability optimization seed generation takes 47% of the total runtime, while wirelengh-driven placement seed generation and post-global placement respectively takes 27% and 26% of the the total runtime. And during the routability optimization, routing analysis uses 6% , window based cell spreading uses 26%, and the rest (such as cell inflation and quadratic programming) uses 15% of the total runtime.

Besides, compared with pure wirelength-driven placer POLAR [33], the proposed routability-driven placer is only 2.26× slower. Considering the fact that routablity-driven placement problem is much more complex than pure wirelength-driven placement, POLAR 2.0 is very fast.

3.5.2 Compared with previous works

The solution quality (including HPWL and ACE [50]) of POLAR 2.0 is shown in Table 3.2. It can be seen that the ACE [50] penalty is decreased to a very low level, which means that POLAR 2.0 can effective immigrate the routing congestion. Fig. 3.4 shows the global placement and routing congestion map of benchmark superblue7 just before/after the routability optimization stage. Fig. 3.4(a) and Fig. 3.4(b) are respectively the global placement just before/after routability optimization. Fig. 3.4(c) and Fig. 3.4(d) are the congestion map of benchmark superblue7 just before/after routability optimization. Note that the congestion value is scaled from -30 to 50, higher value means more congestion. The congestion map

Table 3.2 ACE on ICCAD 2012 benchmarks

Benchmark	ACE(%)				RC	HPWL	sHPWL
	0.50	1.00	2.00	5.00	(%)	($\times 10^7$)	($\times 10^7$)
superblue1	102.48	101.24	100.62	100.25	101.15	2.72	2.82
superblue3	102.29	101.15	100.57	100.23	101.06	3.23	3.33
superblue4	102.08	101.04	100.52	100.21	100.96	2.18	2.24
superblue5	101.51	100.76	100.38	100.15	100.70	3.44	3.51
superblue7	101.77	100.88	100.44	100.18	100.82	3.97	4.07
superblue10	102.40	101.20	100.60	100.24	101.11	6.01	6.21
superblue16	102.81	101.41	100.70	100.28	101.30	2.61	2.72
superblue18	103.17	101.58	100.79	100.32	101.47	1.61	1.69

Table 3.3 Comparison on ICCAD 2012 benchmarks

Benchmark	SimplR [44]		coPR [133]		Ripple2 [43]		NTTPlacer4 [46]		POLAR 2.0	
	sHPWL	Time (s)	sHPWL	Time (s)	sHPWL	Time (s)	sHPWL	Time (s)	sHPWL	Time (s)
superblue1	2.79	2319	2.86	2453	2.89	10213	2.79	8759	2.82	1612
superblue3	3.44	2706	3.46	2603	3.60	15114	3.67	7193	3.33	1760
superblue4	2.43	1257	2.37	1816	2.27	8575	2.31	4866	2.24	1260
superblue5	3.60	2154	3.51	2345	3.49	10833	3.59	7322	3.51	1100
superblue7	4.31	3249	4.36	3570	4.29	23017	3.96	15005	4.07	1863
superblue10	6.91	4837	6.51	5098	5.98	26312	6.17	12352	6.21	2946
superblue16	2.86	1797	2.80	1234	2.84	9494	2.78	6024	2.72	1028
superblue18	1.82	1645	1.68	1342	1.84	10989	1.64	4622	1.69	1278
Normalize	1.051	1.54	1.030	1.56	1.0282	8.83	1.0111	5.22	1.000	1.00

was drew based on the results of FastRoute's pattern routing [136]. It can be seen that the shape of wirelengh-driven placement seed is roughly maintained, while the routing congestion is significant spread out.

As shown in Table 3.3, our placers outperforms other academic routability-driven placers on ICCAD 2012 benchmark suite [51]. Considering the scaled HPWL, on average, our placer respectively achieves 5.1%, 3.0% , 2.8% and 1.1% improvement versus SimPLR [44], coPR [133], Ripple2 [43] and NTUPlace4 [46]. Considering the runtime, on average, we believe POLAR 2.0 is faster than SimPLR, coPR, Ripple2 and NTUPlace4. [1]

[1] For SimPLR [44], Ripple 2.0 [43] and NTUPlace4h [46], their results were referred from the ICCAD 2012 contest [51]; for coPR [133], its runtime was computed based on [133] which claimed that it was 1.01 slower than SimPLR. Besides, the machine used in the ICCAD 12 contest is with Intel Xeon X7560 2.27GHz CPU(much more expensive and powerful than the CPU used in our experimental environment) and 16GB memory. Therefore, the datas in Table III are relatively correct.

3.6 Conclusions

In this chapter, we propose a very simple and fast routability-driven placer, POLAR 2.0, which targets on mitigating routing congestion by the following two basic approaches: (i) minimizing routing demand by maintaining a good wirelength-driven placement; (ii) spreading the routing demand properly by a novel routability-driven rough legalization and a history based cell inflation.

Experimental results show that even without applying many techniques that others proposed (such as narrow channel blocking [46, 24], routing path based inflation [133, 43], and reserving space around macros [135], etc), POLAR 2.0 yet outperforms all published academic routability-driven placers. For future work, we will investigate the use of those techniques.

CHAPTER 4. POLAR 3.0: AN ULTRAFAST GLOBAL PLACEMENT ENGINE

Placement is one of the most important problems in electronic design automation. Although it has been investigated for several decades, a more efficient core engine is critically needed for the following reasons: (i) design scale becomes huge; (ii) placement is typically run again and again to explore the design space at early design stages (e.g., physical synthesis); (iii) placement core engine is called many times to iteratively optimize other objectives (e.g., timing and routability). In this chapter, we propose a new ultrafast global placement engine called POLAR 3.0, which explores parallelism in state-of-the-art quadratic placer. POLAR 3.0 can make full use of multi-core system and it delivers 7-30$\times$ speedup over state-of-the-art academic placers by using a 8-core CPU, while the solution quality is competitive.

4.1 Introduction

Placement is considered to be one of the most important problems in electronic design automation (EDA). Although it has been extensively studied for decades, it is still a very challenging problem and more efficient placers are critically needed. Firstly, the scale of placement is increasing continuously to tens of millions cells nowadays. Secondly, placement is used in early design stages (e.g., physical synthesis) to guide the design process, and it is typically run many times to explore the design space. Last but not least, multiple objectives, such as wirelength, timing and routability, should be optimized simultaneously, and the typical approach is to transform the problem into a sequence of wirelength-driven placement problems. Therefore, [1] emphasizes the importance of developing a high performance placement core engine, which minimizes wirelength.

Problem scale has significant impact on the evolution of global placement core engine. In the early age, simulated annealing based placers (e.g., Timberwolf [7]) perform very well for small design. Then industry switches to min-cut based placement techniques (e.g., Capo [10] and Dragon [8]) for medium design. When design scale arrives at hundreds of thousands cells or even several millions, analytical placers [29, 11, 24, 31, 34, 22, 27, 33, 30, 32, 26] are considered the only effective method . However, when we are talking about huge design which might have hundreds of millions cells, the existing placers are still not fast enough considering modern design flow is iterative and placement should be performed many rounds.

To catch up with continuously increasing design scale, multi-threading has been widely used in EDA industry. However, it is challenging to achieve high parallelism for placement problem. For quadratic placer, such as SimPL [27] and POLAR [33], wirelength is minimized by quadratic programming (QP), which is solved as a sparse symmetric positive definite linear system by a preconditioned conjugate gradient (PCG) method. Wirelength minimization by QP dominates the total runtime of global placement stage. Since the x- and y-directed wirelengths are independent of each other, the two directions of wirelength optimization can be easily parallelized with two threads, one thread for each direction. However, PCG is known to be hard to parallelize [128] due to limited memory bandwidth and data dependency. Speedup does not scale well with more CPU cores.

For nonlinear placers [29, 24, 26, 34, 32], nonlinear programming consumes most of CPU runtime. In nonlinear placer, all the constraints are transformed into penalty functions. As a result, the cost function is not decomposable into two independent functions as in quadratic placers. Therefore, it is even more difficult to parallelize nonlinear placers.

In this chapter, we systematically study the performance bottleneck of parallelism in quadratic placer. We propose an ultrafast global placement engine called POLAR 3.0. To achieve high scalability that previous works have not reached, the global placement iterations are divided into a series of *frames*, in which partitioning is applied based on cells' locations and then placement of each partition is performed simultaneously. To reduce loss of solution quality, *frames* are configured to allow varied partitioning, in order to prevent cells from being restricted in the same partition. Experimental results show that POLAR 3.0 can make full use of multi-core

system and it delivers 7-30× speedup over state-of-the-art academic placers with competitive solution quality by using a 8-core CPU. The main contributions of this chapter are concluded as follows.

- We systematically study the performance of state-of-the-art quadratic placers and point out that parallelizing global placement with high scalability is a very challenging rather than simple problem.
- We propose a new global placer to make full use of multi-core system. It is almost one order of magnitude faster than state-of-the-art academic placers, with competitive solution quality.

The rest of this chapter is organized as follows. Section 4.2 is a preliminary to global placement. In Section 4.3, we point out the inherent challenges of parallelizing global placement with high scalability. In Section 4.4, we illustrate the ultrafast global placement engine POLAR 3.0. Experimental results are presented in Section 4.5. Finally, in the Section 4.6, we make conclusions.

4.2 Preliminary

In placement problem, a circuit can be represented by a hypergraph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_{|V|}\}$ is the set of cells and $E = \{e_1, e_2, \ldots, e_{|E|}\}$ is the set of nets. Global placement tries to determine physical positions of cells without violating density constraints. We denote the x-coordinates of cells by a vector $\mathbf{x} = (x_1, x_2, \cdots, x_{|V|})$, and the y-coordinates by $\mathbf{y} = (y_1, y_2, \cdots, y_{|V|})$. The objective is the half perimeter wirelength (HPWL), which is measured by Formula 4.1.

$$\text{HPWL}(\mathbf{x}, \mathbf{y}) = \Sigma_{e \in E}[\max_{i \in e} x_i - \min_{i \in e} x_i + \max_{i \in e} y_i - \min_{i \in e} y_i] \tag{4.1}$$

4.2.1 Quadratic placement

In quadratic placer, a multi-pin net can be decomposed into a set of two-pin nets by bound2boubd (B2B) net model [31]. The Manhattan distance of two connected pins is ap-

proximated by their squared Euclidean distance, so the cost function ϕ of global placement can be defined in Formula 4.2.

$$\phi = \frac{1}{2}\mathbf{x}^T Q_x \mathbf{x} + \mathbf{c}_\mathbf{x}^T \mathbf{x} + \frac{1}{2}\mathbf{y}^T Q_y \mathbf{y} + \mathbf{c}_\mathbf{y}^T \mathbf{y} + const \tag{4.2}$$

where the connection matrices Q_x and Q_y are both sparse symmetric positive definite. Minimizing ϕ is equal to solving the linear systems 4.3 and 4.4.

$$Q_x \mathbf{x} = -\mathbf{c}_\mathbf{x} \tag{4.3}$$

$$Q_y \mathbf{y} = -\mathbf{c}_\mathbf{y} \tag{4.4}$$

To reduce cell overlapping, [116] proposed a simple way to add spreading force by pseudo net connecting cell's original position to its anchor (i.e., desirable location). Then the linear systems are updated and solved again.

Since quadratic placement formulation was first proposed, there are many improvements in academic quadratic placers. The most recent progress is a spreading approach called rough (look-ahead) legalization [27]. Many placers [27, 28, 42, 33, 45] based on rough legalization produce competitive results on the placement contests [124, 125, 52, 50, 51].

4.3 Challenge Of Parallelization

Parallelizing state-of-the-art quadratic global placers, such as SimPL [27], to achieve high scalability is a very challenging rather than simple problem. The common global placement framework with rough legalization [27] is presented in Fig. 4.1. The three most time consuming components are respectively linear system generation by B2B model, solving linear system by PCG and rough legalization.

Firstly, linear system generation is difficult to parallelize. Without loss of generality, let us look into how to generate linear system 4.3 for x-direction. Compressed row storage (CRS) [128] is the most efficient format to store the non-zero elements of sparse matrix in our application. Since Q_x is symmetric, we only need to store its lower triangular part. For any non-zero element

$Q_x[i][j]$ $(i \geq y)$, multiple nets may contribute to it. For example, suppose there are two-pin nets N_1 and N_2 connecting movable cells C_1 and C_2, N_1 and N_2 contribute a and b to the non-zero element $Q_x[2][1]$ respectively, then $Q_x[2][1] = a + b$. Therefore, nets have to be scanned one by one to generate all the non-zero elements [1]. Since linear systems 4.3 and 4.4 are independent of each other, two threads can are used for generating them, one is for 4.3 and the other is for 4.4. However, this is all where we can apply parallelism. The experiments show that SimPL [143] can only achieve 1.86× speedup in this step by using 8 threads. The theoretical speed cannot be more than 2× speedup.

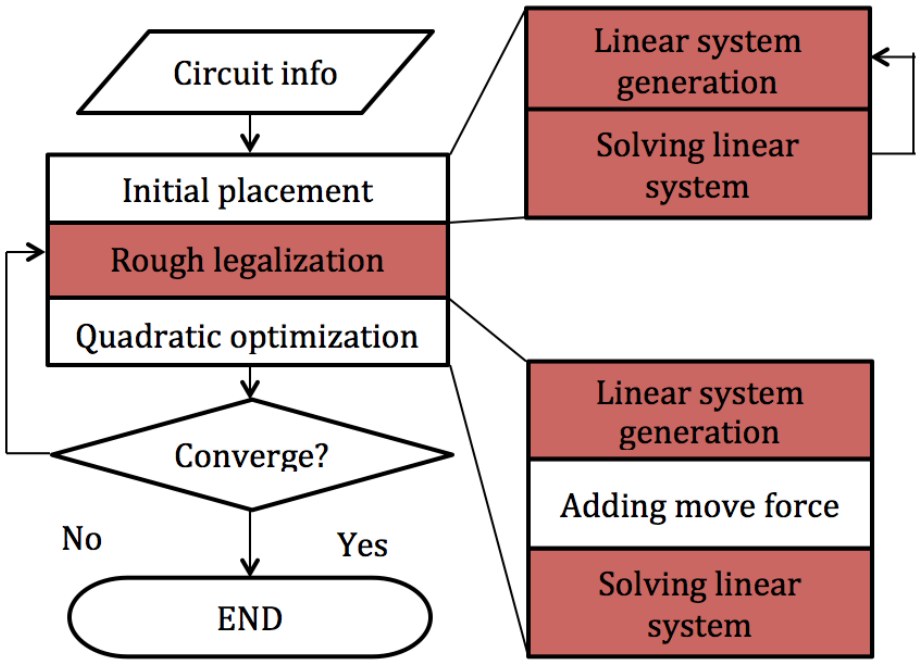


Figure 4.1 The global placement framework with rough legalization. The three most time consuming components are highlighted.

Secondly, rough legalization is intrinsically sequential, since density hotspots are dealt with one by one. For each hotspot, it should get through two steps: expansion region search and cell spreading in expansion region. The runtime bottleneck is the second step, and the runtime

[1] Another method is to add lock for each non-zero element. Although nets can be scanned parallelly, the overhead of adding/releasing locks would overwhelm that benifet of parallelism and make the program extremely slow.

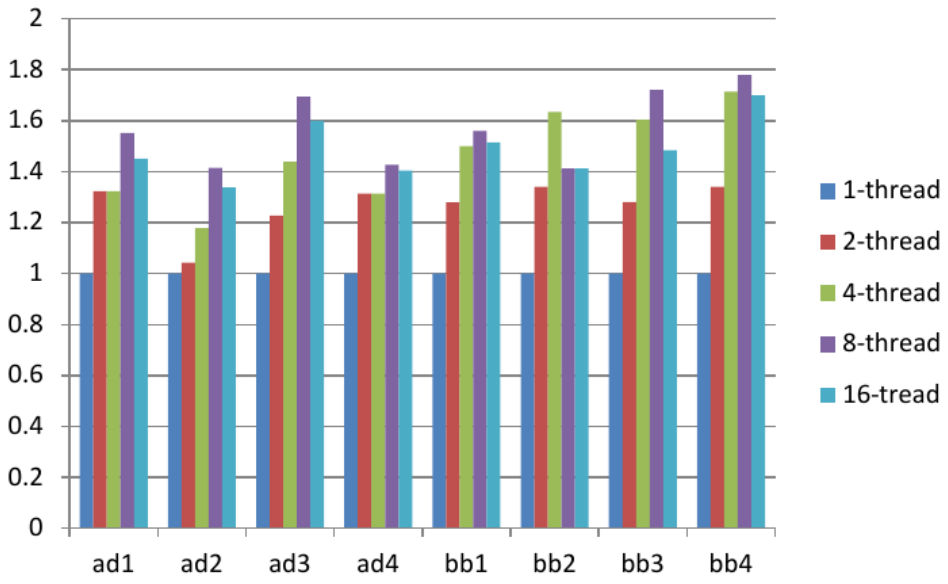


Figure 4.2 Parallelizing Jacobi PCG solver in Intel MKL library in quadratic placement application. The y-axis is speedup.

of the first step can be ignored compared with that of the second step. Although parallelism is applied in the second step, [143] shows that SimPL only achieves 1.62× speedup in rough legalization by using eight threads. For POLAR [33], parallelizing the second step is even harder since it has already been highly optimized for runtime by lazy update technique. Table 4.1 gives the average runtime of rough legalization per global placement iteration for SimPL and POLAR, runtime is measured in seconds. The experiment was performed on the same machine over ISPD2005 benchmarks [124].

Table 4.1 Average runtime of rough legalization per global placement iteration

benchmark	SimPL	POLAR
adaptec1	0.52	0.12
adaptec2	0.61	0.16
adaptec3	1.87	0.31
adaptec4	1.57	0.36
bigblue1	1.79	0.21
bigblue2	1.65	0.28
bigblue3	3.79	1.01
bigblue4	9.87	1.72
Norm.	1.00	0.21

Thirdly, PCG is widely used in science and engineering and there is still not effective algorithm to parallelize it so far [128]. To verify this, the performance of cutting edge PCG solver in Intel MKL library [144] was measured by using a Intel Xeon E5-2640 v3 CPU, which has 8 cores. We used B2B net model to generate linear systems over ISPD2005 benchmarks [124], and then solved them by using different number of threads. The experimental results are presented in Fig. 4.2. It shows that the speedup is less than 2× and is increasing extremely slowly as the number of threads is increased. Note that, Running PCG solver with 16 threads is even a little slower than that with 8 threads in the experiment. That is because a core can launch 2 hyper-threads and these two hyper-threads share the same execution resources, such as L1 and L2 cache, and are not truly parallel.

To further demonstrate the challenge by experiment, we implemented POLAR [33] and parallelized it in the following way using OpenMP: (i) parallelize every loop that can be parallelized; (ii) all the computation that related to x-direction is parallel to that of y-direction. We set the number of threads to 1, 2, 4, 8 and 16 respectively, and ran POLAR over ISPD2005 benchmarks. Fig. 4.3 shows that we only get less than 1.8× speedup. Note that, launching more threads even slows down the program over some test cases. This is not abnormal for complex applications, for example, the ones whose bottleneck is memory bandwidth.

To sum up, existing methods [27, 33] has not reached to 2× speedup, and how to break through this limit while maintain good solution quality is the major challenge of this work.

4.4 POLAR 3.0

To resolve the above challenge, we resort to an ancient while powerful strategy–divide and conquer. Divide and conquer used to be applied in global placement, such as partitioning based methods [12, 13, 14, 15, 16]. However, since analytical techniques were proposed, partitioning based approaches are considered inferior. In POLAR 3.0, we show that partitioning is not outdated and can be leveraged to speed up analytical placer without sacrificing solution quality significantly.

We chose POLAR as our base engine and built up a new global placer on it. The basic idea is to divide global placement iterations into a series of *frames*, in which partitioning is applied

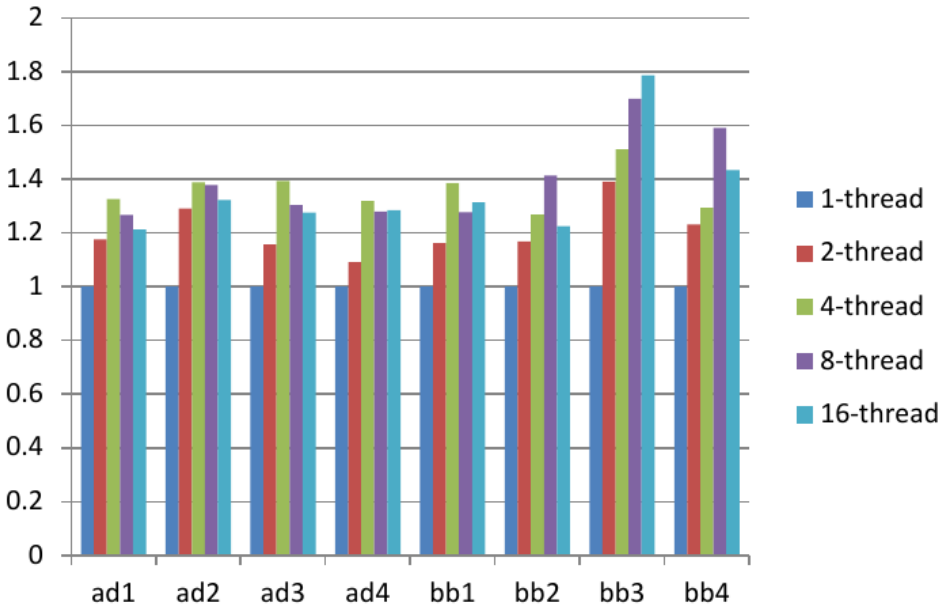


Figure 4.3 Run POLAR by multi-threading. The y-axis is speedup.

based on cells' locations and placement of each partition can be performed independently by multi-threading.

4.4.1 Framework

The framework of POLAR 3.0 is presented in Fig. 4.4. The first stage is initial placement, and then rough legalization. In the next stage, global placement iterations are divided into a series of *frames*. Each *frame* begins with a roughly legalized placement. Partitioning is applied based on cells' locations, and then placement of each partition (also called sub-placement in the rest of chapter) is performed independently by multi-threading. Each sub-placement goes through several iterations of four processes: linear system generation, adding move force, solving linear system and rough legalization. Once all the sub-placements finish, POLAR 3.0 enters into synchronization, where the locations of all the cells are updated. POLAR 3.0 continues until the wirelength is not improved.

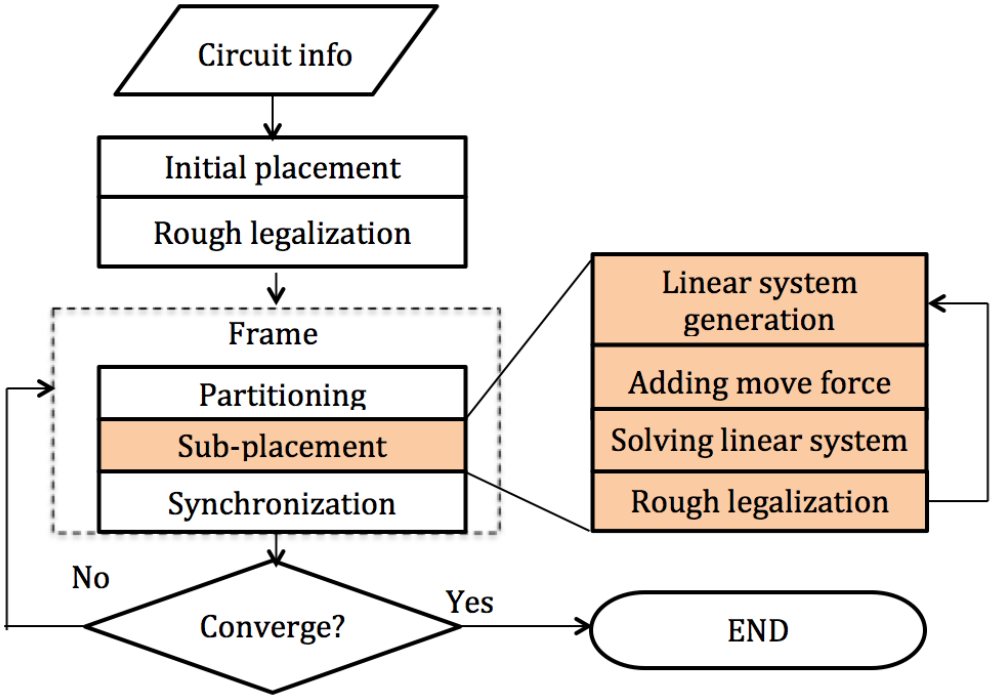
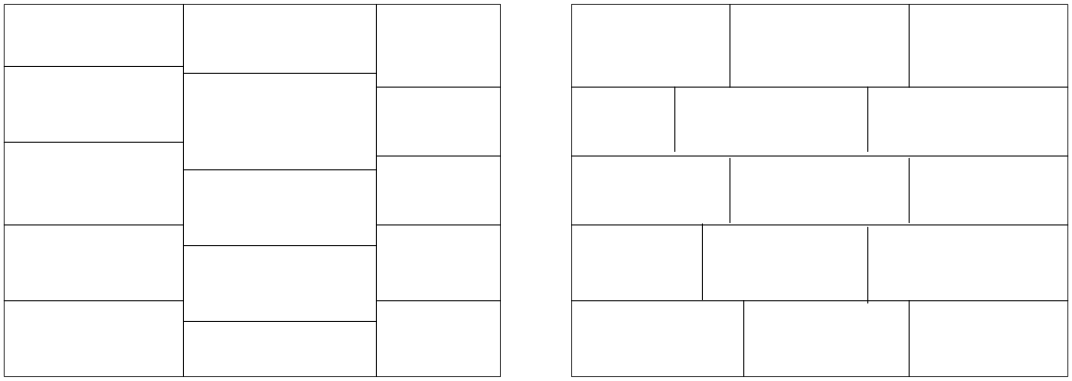


Figure 4.4 The global placement framework of POLAR 3.0.

4.4.2 Placement-driven partitioning

In each *frame*, POLAR 3.0 starts with a roughly legalized placement. Partitioning is performed based on cells' locations. Therefore, we call it placement-driven partitioning. Different from traditional partitioning based approaches, connection information (i.e., netlist) is never used. The goal of placement-driven partitioning is to divide the whole circuit into many small partitions with similar size, while traditional methods, such as hMETIS [18], try to minimize the number of connections among different partitions.

The whole circuit is divided into a set of partitions by horizontal and vertical cut lines. A partitioning scheme is represented by a tuple (xx, yy, dd), where $xx - 1$ and $yy - 1$ are respectively the number of horizontal and vertical cut lines. If dd is 0, POLAR 3.0 applies horizontal cut first and then vertical cut. On the contrary, if dd is 1, vertical cut is applied before horizontal cut. Fig. 4.5 gives two instances of partitioning schemes, vertical cut is applied firstly in Fig. 4.5(a), while horizontal cut is applied firstly in Fig. 4.5(b).

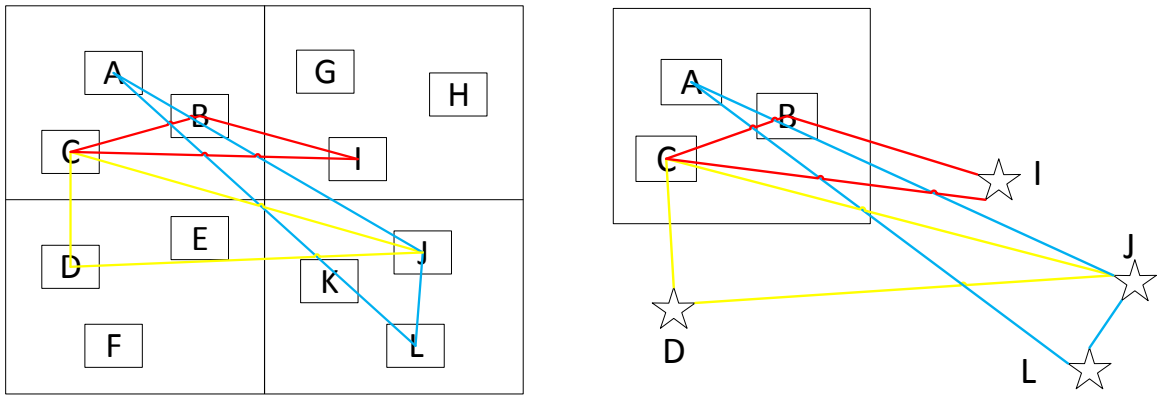


(a) Scheme: $(5, 3, 1)$ (b) Scheme: $(5, 3, 0)$

Figure 4.5 Two partitioning schemes.

Each partition is composed of a region, a set of cells and a set of nets, so it can be considered as a small instance of circuit. For each partition, if a net connects at least one cell inside of it, this net is kept in (belongs to) this partition. Besides, pins are considered fixated at their current locations if they are outside of partition. For example, as shown in Fig. 4.6, the whole circuit is divided into four partitions. The top-left partition contains three cells A, B and C. There are three nets associated with at least one of A, B and C. For the red net, it connects to cell I, which is outside of top-left partition, so cell I is considered fixated for the top-left partition. By above setting, placements of partitions are independent of each other, and each sub-placement is an placement instance. Note that, once all the sub-placements finish, all the cells' positions are updated before going to next *frame*.

To maximize parallelism, each partition is expected to have a similar number of cells. Otherwise, the placement of the partition which has the most number of cells would become the runtime bottleneck in all the sub-placements. Algorithm 11 gives the method of placement-driven partitioning. The placement region is split into a set of uniformed bins by a $m \times n$ grids. The number of cells in each bin is calculated, and then a lookup table can be built to quickly return the number of cells in any rectangular region which is composed of bins. When a



(a) The whole circuit (b) Top-left partition

Figure 4.6 Pins are considered fixated at their current locations if they are outside of partition.

rectangular region (e.g., the whole placement) is partitioned, the locations of cut lines can be easily computed by Algorithm 12[2].

Algorithm 11 Placement-driven partitioning

Require: Placement is split into a set of uniformed bins.
Require: A roughly legalized placement and a partitioning scheme (xx, yy, dd).
Require: Horizontal (or vertical) cut line should be exactly one of the lines that are used to split placement into bins.
Ensure: Each partition has similar number of cells.
1: Build up a lookup table to quickly return the number of cells in any rectangular region;
2: Apply horizontal (or vertical) cut first if $dd = 0$ or 1;
3: **for** any rectangular region produced in the last step **do**
4: Apply vertical (or horizontal) cut if $dd = 0$ or 1;
5: **end for**

4.4.3 Varied partitioning scheme

If placement-driven partitioning is only applied once and then sub-placements are performed until the end of global placement stage, it would result in significant quality loss because each cell is restricted in the same partition and cannot be migrated from one partition to another, which leads to limitation of solution space. To resolve this issue, we introduce the concept of varied partitioning scheme.

[2]Without loss of generality, Algorithm 12 presents how to compute the locations of horizontal cuts. For vertical cuts, the method is similar.

Algorithm 12 Find the locations of cut lines

Require: A rectangular region represented by (l_x, l_y, r_x, r_y) in bin coordinate system, where (l_x, l_y) and (r_x, r_y) are respectively the coordinates of its left-bottom and right-up bins.
Require: : The number of horizontal cut lines, denoted by k.
Ensure: The locations of horizontal cut lines, stored in an array $p[1..k]$.

```
Initialize an array n[l_y..r_y] = 0;
for i = l_y to r_y do
    n[i] = the number of cells in rectangular region (l_x, l_y, r_x, i);
end for
m = n[r_y]/(k+1);
for i = 1 to k do
    find the index l and u, that satisfy n[l] ≤ m * i ≤ n[u];
    if m * i − n[l] < n[u] − m * i then
        p[i] = l;
    else
        p[i] = u;
    end if
end for
```

A *frame* is composed of a partitioning scheme and several global placement iterations, so it can be represented as a tuple (xx, yy, dd, n), while (xx, yy, dd) is the scheme of placement-driven partitioning and n is the number of global placement iterations applied in the *frame*. To prevent cells from being restricted in the same partition, the *frame* configuration is varying from one frame to next. For example, if *frame* $(4, 4, 0, 5)$ is used first and then $(1, 1, 0, 1)$, it means that we apply a 4×4 placement-driven partitioning to get 16 partitions and perform 16 sub-placements in the next 5 global placement iterations, and then one flat global placement iteration is performed to allow cells to move outside of their partitions, in order to get better locations.

The design of *frame* configuration is important for runtime performance and solution quality. For example, If all the *frames* are configured to $(1, 1, 0, 1)$, then POLAR 3.0 is degenerated to POLAR [33] and cannot leverage multi-core system. However, if all the *frames* are configured to $(5, 5, 0, 1)$, then placement quality is significantly suffered. Therefore, the design of *frame* configuration is a trade-off between runtime performance and placement quality. In Section 4.5, we will further discuss about how to choose proper configuration of *frames* in the experiments.

4.4.4 Support partitioning efficiently

To make sub-placements run independently, a straight forward idea is to reconstruct the data structures (such as nets and cells) for each partition. It is the initial implementation of POLAR 3.0. However, this idea has several performance issues.

Firstly, memory usage is increased as the number of partitions is increased. For one thing, the memory to store a cell is doubled. One copy is for the whole circuit, and the other is for the partition to which this cell belongs. For another thing, which is even worse, a net (contains a set of pins) which crosses over k partitions has k copies, since each partition has one copy. Overall all, the whole memory usage is increased by several times. Huge memory footprint not only limits the scalability, but also slows down the program (e.g., results in higher cache miss rate). Besides, we observed that reconstructing netlist for each partition is relatively time consuming (about 20% of total runtime in the initial implementation).

To address the above issues, we redesigned the architecture of POLAR [33] in order to support partitioning efficiently. The key idea is to let partition share memory from the whole circuit. Therefore, there is only one copy for each cell/net/pin in the memory. By this new architecture, the runtime of partitioning is reduced to less than 1% of total runtime in the final implementation of POLAR 3.0, and can support partitioning as many as possible.

Fig. 4.7 presents the memory footprints of the whole circuit and the four partitions for the instance in Fig. 4.6. In the whole circuit, as shown in the top table of Fig. 4.7, we have a cell list, net list and pin list. Cells, nets and pins are stored in different kind of structures maintaining their own variety of information (e.g., width, height and location for cell, weight for net, offset (or cell id that it may have) for pin). Each net has a pair of values to store the start and end index of its pins in the pin list. In the partition, for each cell (or net), which belongs to this partition, only its id rather than its structure is stored. A hash table is used to map cell id in the whole circuit to a new id in the partition. This hash table has two functions: (1) check whether a cell belongs to this partition; (2) used in linear system generation, the new id of cell in its partition is exactly the same as its row index in the matrix Q_x (Q_y). Besides, the boundary of placement region should be stored for the whole circuit and partitions.

Scanning netlist on circuit (or partition) is the most fundamental operation in placement algorithm. It is the basic to implement a placer and widely used in many places, such as generating linear system and calculating total wirelength. Algorithm 13 shows how to scan netlist on partition. The hash table is used to check the condition in line 6.

Algorithm 13 Scan netlist on a partition

```
1:  for any net id belongs to the partition do
2:      fetch the corresponding net structure;
3:      get the start and end index of its pins in the pin list, denoted by s and e;
4:      for i = s; i ≤ e; i = i + 1 do
5:          fetch the corresponding pin structure;
6:          if this pin belongs to some cell in the partition then
7:              //it is a movable pin;
8:          else
9:              //it is a fixed pin;
10:         end if
11:     end for
12: end for
```

4.5 Experimental Results

POLAR 3.0 is implemented in C++, and OpenMP is used to support parallelism. To verify the efficiency of POLAR 3.0, we obtained binaries of some state-of-the-art academic placers, such as FastPlace [22], NTUplace3 [26], ComPLx [28] and ePlace [34], and ran them over the same benchmarks on the same platform.

The machine that was used in the experiments is a x86_64 Linux server, which has a Intel(R) Xeon(R) E5-2640 v3 CPU (with 8 cores, 2.6GHz frequency and 20M cache) and 132 GB RAM.

The benchmarks were downloaded from the official websites of some contests, including ISPD2005 [124], ISPD2006 [125], ISPD2011 [52], DAC2012 [50][3]. For those benchmarks from routability-driven placement contests [52, 50], routing information is ignored since all the above placers are wirelength-driven. The benchmarks from recent ISPD detailed routing-driven placement contests [53, 145] are not used in the experiments, because (i) all the above placers do not have an interface to read lef/def format files; (ii) those large benchmarks are generated based

[3]The benchmarks of ICCAD2012 contest [51] are the same as those of ISPD2011 and DAC2012 contest.

on previous contests [52, 50, 51]. Although POLAR 3.0 supports users to set target density, the target density is set to 1 for all the benchmarks based on the following two considerations: (i) only ISPD2006 contest benchmarks are specified target density; (ii) Specifying arbitrary target density does not make sense, since the placement density is significantly influenced by other factors, such as routability and timing.

In the experiments, we use the same *frame* configuration for all the benchmarks. As mentioned in Section 4.3, the design of *frame* configuration is to trade-off runtime and solution quality. If many partitions are used in the previous *frame*, then there are fewer partitions in the next frame to make up quality loss. Besides, flat placement is tried to avoid as much as possible, since it cannot make full use of multi-core system. We tried several *frame* configurations and picked a pretty good one for experiments. In this configuration, every four *frames* $(3, 5, 0, 5)$, $(1, 1, 0, 1)$, $(5, 3, 1, 5)$ and $(1, 1, 0, 1)$ are defined as a group and applied repeatedly. Each sub-placement would use at most two threads, for example, when generating (or solving) linear systems 4.3 and 4.4 simultaneously. Therefore, POLAR 3.0 would launch at most 30 threads. Note that this configuration of *frames* does not necessarily give the best average results for all the benchmarks. Besides, the CPU used in the experiments only has 8 cores. Theoretically speaking, POLAR 3.0 could get more speedup by using a CPU which has more cores[4].

The experimental results are presented in Table 4.2, where runtime is measured in seconds. All the detailed placement are performed by the detailed placer inside of NTUplace3. We reported the wirelength and runtime of both global and detailed placement for each placer. "G-WL" and "G-RT" represent the wirelength and runtime of global placement, while "D-WL" and "D-RT" represent the wirelength and runtime of detailed placement. FastPlace crashes on some benchmarks, which are annotated "crash". ePlace does not converge on some benchmarks, which are annotated "fail".

On average, compared with ePlace, POLAR 3.0 is about 4% worst on wirelength, while 31.8× faster. Compared with FastPlace, NTUplace3 and ComPLx, POLAR 3.0 is at least not worse or even better on solution quality, while significantly faster. The average wirelength ratio

[4]We currently do not have 16-core or 32-core CPU to verify this, but will do it later.

among FastPlace, NTUplace3, ComPLx and POLAR 3.0 is 1.07:1.04:1.00:1.00, and runtime ratio is 6.88:32.9:7.55:1.00.

For all these placers, the runtime of detailed placement is roughly comparable. To previous academic placers (not only FastPlace, NTUplace3, ComPLx and ePlace), global placement stage is more time consuming than detailed placement stage. Besides, global placement stage has more impact on solution quality compared with detailed placement stage. Therefore, much less works pay attention to detailed placement. However, for POLAR 3.0, on the contrary, detailed placement stage uses much more runtime than global placement stage. From runtime perspective, we hope that POLAR 3.0 would inspire more research on detailed placement. Besides, we believe that detailed placement stage is relatively easy to parallelize compared with global placement stage[5]. Therefore, we are confident that it can achieve similar speedup combing global and detailed placement stage and we will parallelize detailed placement stage in our future work.

Table 4.2 Comparison with the state-of-the-art academic placers

benchmark	test case	FastPlace				NTUplace3				ComPLx				ePlace				POLAR 3.0			
		G-WL	G-RT	D-WL	D-RT	G-WL	G-RT	D-WL	D-RT	G-WL	G-RT	D-WL	D-RT	G-WL	G-RT	D-WL	D-RT	G-WL	G-RT	D-WL	D-RT
ISPD05	adaptec1	80.6	73	79.2	44	81.5	217	80.3	33	80.2	92	78.1	40	73.1	190	75.6	82	78.6	12	78.9	34
	adaptec2	94.7	107	93.6	64	91.4	245	90.2	49	90.7	103	90.0	53	83.5	250	84.9	46	86.5	15	86.8	48
	adaptec3	214.5	209	215.5	112	239.8	570	233.8	105	206.0	261	206.2	96	193.9	906	196.5	82	206.1	26	207.2	92
	adaptec4	201.9	208	198.7	131	222.5	689	215.0	134	186.5	221	184.3	113	178.1	956	179.0	94	187.0	28	188.6	95
	bigblue1	100.7	103	97.6	70	96.2	432	98.7	49	96.2	175	94.3	51	89.6	298	91.0	41	95.0	16	95.2	48
	bigblue2	160.4	205	155.7	177	162.9	998	158.3	163	147.7	242	145.3	159	139.8	504	142.0	137	144.1	29	145.3	142
	bigblue3	371.8	487	373.4	329	351.3	1100	346.3	208	326.4	486	334.7	271	299.5	1432	308.2	207	324.2	64	329.6	340
	bigblue4	855.2	1024	840.6	663	852.2	3233	829.1	539	796.1	1379	788.9	542	738.6	3682	752.8	416	805.7	160	808.0	492
ISPD2006	adaptec5	329.0	473	366.6	405	345.8	1238	344.6	177	324.0	428	322.4	211	294.8	1074	301.1	171	327.1	57	329.6	198
	newblue2	193.4	183	203.4	119	188.4	627	191.7	75	185.9	264	189.6	98	171.5	361	184.3	91	181.4	25	189.4	102
	newblue3	298.8	185	293.1	177	284.4	505	275.9	175	265.9	216	261.4	153	259.4	585	262.6	134	266.7	20	264.8	162
	newblue4	254.7	270	250.7	171	252.2	1036	247.6	149	237.1	328	232.9	138	216.1	882	222.9	114	238.2	38	239.0	127
	newblue5	424.7	616	472.4	481	429.1	2080	426.9	299	411.6	707	406.2	331	372.0	1578	383.1	257	410.4	88	415.6	289
	newblue6	516.1	595	506.3	376	505.0	1936	498.2	330	477.0	736	470.8	323	433.0	2136	443.1	255	478.1	88	479.6	291
	newblue7	1086.4	1063	1072	958	1114.4	3886	1100.2	525	998.1	1907	989.8	776	937.8	2612	956.9	648	984.6	187	990.8	683
ISPD11	superblue1	292.4	280	287.1	328	271.9	1620	267.9	180	259.5	261	256.5	248	269	6589	250.3	178	253.2	48	253.5	221
	superblue2	crash	crash	crash	crash	615.0	2778	609.7	211	605.9	323	608.2	382	fail	fail	fail	fail	592.8	52	592.5	273
	superblue4	crash	crash	crash	crash	217.8	757	216.5	121	214.4	187	212.3	152	fail	fail	fail	fail	209.0	26	209.3	132
	superblue5	358.6	242	355.1	237	352.9	1391	345.1	181	349.2	229	337.7	219	fail	fail	fail	fail	338.8	37	336.5	184
	superblue10	crash	crash	crash	crash	557.7	1356	551.2	215	538.3	377	535.6	246	520.9	1644	527.8	204	531.6	54	534.8	212
	superblue12	277.1	543	271.4	601	241.6	5804	238.7	333	242.9	642	237.8	479	206.5	3158	212.1	291	240.5	98	238.7	370
	superblue15	313.5	264	310.1	240	295.8	2124	297.3	155	297.2	385	295.0	191	280.9	892	283.9	175	292.2	51	294.8	188
	superblue18	157.0	190	152.2	169	139.1	1414	139.9	114	141.5	169	137.7	147	137.9	641	136.9	118	137.4	27	137.3	128
DAC12	superblue3	322.4	324	317.5	351	309.3	1493	302.9	219	314.1	267	304.8	282	fail	fail	fail	fail	299.0	45	300.1	228
	superblue6	crash	crash	crash	crash	318.6	1819	319.8	209	319.5	358	318.5	291	fail	fail	fail	fail	313.4	53	315.4	247
	superblue7	404.7	541	395.6	422	379.8	3467	377.3	294	388.6	530	385.0	395	366.1	1788	368.6	301	375.7	83	375.7	318
	superblue9	239.2	308	236.1	278	222.0	2285	221.8	204	219.7	305	217.2	233	fail	fail	fail	fail	214.6	46	215.2	219
	superblue11	crash	crash	crash	crash	335.9	1465	335.7	179	334.9	277	337.4	202	774.6	5399	391.5	623	338.5	48	337.4	199
	superblue14	236.4	197	234.7	202	224.2	1310	229.9	136	222.2	197	220.4	165	fail	fail	fail	fail	218.6	28	219.1	156
	superblue16	269.1	208	271.4	223	259.1	1662	262.8	120	253.9	192	256.7	190	fail	fail	fail	fail	260.4	37	258.7	144
	superblue19	154.1	204	155.8	216	143.9	1587	144.8	157	147.1	152	147.8	181	fail	fail	fail	fail	145.7	25	145.3	164
Norm.		1.08	6.88	1.07	1.37	1.05	32.9	1.04	0.95	1.01	7.55	1.00	1.15	1.01	31.8	0.96	1.05	1.00	1.00	1.00	1.00

[5]Take FastDP [103] for example, global swap can be parallelized using divide and conquer by restricting swapping scope. Parallelizing row compact is even simple, since every row is independent of others.

4.5.1 POLAR 3.0 runtime analysis

We analyzed the runtime of POLAR 3.0 by using different number of threads. The method is to change the number of partitions in *frame*. As shown in Table 4.3, for example, to measure the runtime of POLAR 3.0 using 8 threads, we can repeatedly apply *frame* group $(2, 2, 0, 5), (1, 1, 0, 1), (2, 2, 1, 5)$ and $(1, 1, 0, 1)$. POLAR 3.0 would launch maximally 8 threads, since each partition would use 2 threads for generating (or solving) linear systems 4.3 and 4.4 simultaneously. The experimental results are shown in Table 4.4, where "WL" is the wirelength of detailed placement and "G-RT" is the CPU runtime of POLAR 3.0 in global placement stage, all the runtime is measured in seconds. To measure the runtime of 1-thread and 2-thread, placement-driven partitioning is not applied. The difference between 1-thread and 2-thread is that generating (or solving) linear systems 4.3 and 4.4 are parallel (one thread for each direction) in 2-thread. Besides, we also added a group of *frames* called "speedy". We can use speedy to gain more speedup at the cost of sacrificing solution quality.

Table 4.3 Frame configuration to measure runtime of POLAR 3.0 by using different number of threads

# of threads	Configuration of *frame* group			
1-thread	(1,1,0,1)	(1,1,0,1)	(1,1,0,1)	(1,1,0,1)
2-thread	(1,1,0,1)	(1,1,0,1)	(1,1,0,1)	(1,1,0,1)
4-thread	(2,1,0,5)	(1,1,0,1)	(1,2,1,5)	(1,1,0,1)
8-thread	(2,2,0,5)	(1,1,0,1)	(2,2,1,5)	(1,1,0,1)
16-thread	(2,4,0,5)	(1,1,0,1)	(4,2,1,5)	(1,1,0,1)
speedy	(3,5,0,5)	(5,3,1,5)	(3,5,0,5)	(5,3,1,5)
default	(3,5,0,5)	(1,1,0,1)	(5,3,1,5)	(1,1,0,1)

By using 2/4/8/16/30 threads, POLAR 3.0 can achieve similar solution quality with POLAR, while gain 1.43/1.59/2.5/4.0× runtime speedup. Note that compared with 2-thread, 4-thread only gains little speedup. The main reason is that the number of nets kept in each partition is very close to that of whole circuit, so the runtime of generating (or solving) linear systems is also very closed to that of the whole placement.

Table 4.4 Comparison of POLAR 3.0 with different number of threads

benchmark	test case	1-thread		2-thread		4-thread		8-thread		16-thread		default		speedy	
		WL	G-RT	WL	G-RT	WL	G-RT	WL	G-RT	WL	G-RT	WL	G-RT	WL	G-RT
ISPD05	adaptec1	76.9	59	76.9	42	77.7	32	79	20	79.1	14	78.9	12	85.9	9
	adaptec2	86.5	72	86.5	51	87.5	39	86.2	24	86.8	17	86.8	15	91.1	10
	adaptec3	200.1	128	200.1	89	199.3	63	202.7	41	204.8	33	207.2	26	241.5	20
	adaptec4	181.6	129	181.6	88	184.9	64	185.4	40	188.4	30	188.6	28	201.2	18
	bigblue1	95.7	80	95.7	56	92.9	42	93.7	26	95.4	20	95.2	16	106.1	11
	bigblue2	143.3	133	143.3	93	143.9	73	144.9	44	145.6	32	145.3	29	147.5	18
	bigblue3	320.4	311	320.4	220	322.9	171	326.2	112	326.1	73	329.6	64	362.5	42
	bigblue4	787.7	704	787.7	508	795.3	426	800.5	260	811.4	188	808	160	835.4	108
ISPD2006	adaptec5	313	258	313	181	320	154	319.6	98	324.6	65	329.6	57	363.2	40
	newblue2	189.2	123	189.2	87	188	65	187.8	37	187.1	29	189.4	25	193.1	16
	newblue3	262.7	93	262.7	63	263.3	51	264.3	32	265.5	25	264.8	20	267.6	15
	newblue4	231.7	184	231.7	132	228.8	104	234	61	233.4	46	239	38	256.9	28
	newblue5	407.7	407	407.7	298	404.5	225	406.9	149	410.6	107	415.6	88	432.6	64
	newblue6	473	399	473	282	469.9	229	471.6	147	477.3	105	479.6	88	498.7	60
	newblue7	986.8	841	986.8	603	982.1	472	978.6	300	980.4	223	990.8	187	1000.9	130
ISPD11	superblue1	251.3	208	251.3	144	249.3	114	251.5	74	251.5	49	253.5	48	258.7	30
	superblue2	585	215	585	149	586.1	119	587.6	77	586.6	56	592.5	52	593.8	36
	superblue4	207.5	114	207.5	79	205.6	58	207.9	42	208.3	31	209.3	26	214.8	17
	superblue5	333.1	160	333	111	334.2	84	333.4	55	335.7	39	336.5	37	340.9	24
	superblue10	534.7	220	534.7	149	532.6	132	533.6	87	534.1	61	534.8	54	540.1	38
	superblue12	225.5	484	225.5	332	229.4	230	232.4	154	235.8	114	238.7	98	258.7	68
	superblue15	294.7	210	294.7	145	290	134	291.1	82	295.8	61	294.8	51	311.7	38
	superblue18	135.9	119	135.9	84	135.2	66	135.1	40	135.4	32	137.3	27	162.3	20
DAC12	superblue3	296.6	207	296.6	145	292.4	124	297.4	77	299.8	57	300.1	45	308.9	32
	superblue6	311	229	311	160	311.1	142	313.2	88	312.2	62	315.4	53	320.6	39
	superblue7	372.3	352	372.3	248	373.4	226	377.8	139	386.3	97	375.7	83	392.2	59
	superblue9	210.6	207	210.6	147	213.9	123	221.1	78	221.6	55	215.2	46	236.6	32
	superblue11	337.9	207	337.9	143	336.4	123	335.5	76	337	50	337.4	48	349.9	33
	superblue14	219.1	135	219.1	94	219.9	83	218.9	49	218.2	35	219.1	28	222.9	21
	superblue16	255.3	164	255.3	118	255.1	96	257.2	60	257.7	43	258.7	37	271.1	26
	superblue19	145.1	120	145.1	83	145.2	63	145.7	43	147.2	31	145.3	25	150.9	17
Norm.		1.000	1.00	1.000	0.70	1.000	0.63	1.006	0.4	1.012	0.25	1.015	0.22	1.069	0.15

4.6 Conclusions

In this chapter, we systematically study the problem of parallelizing state-of-the-art quadratic placer. The main challenge is that the major runtime components, such as solving quadratic problem, are difficult to parallel. Experiments show that existing method can only achieve less than 1.8× speedup by using 16 threads provided by a modern 8-core CPU.

To resolve this challenging problem, we built up a new global placer (POLAR 3.0) based on POLAR to fully leverage multi-core system. In POLAR 3.0, we propose placement-driven partitioning and verify partitioning scheme to trade-off runtime and solution quality. We demonstrate that by reasonable designing of *frames*, POLAR 3.0 could achieve competitive solution quality while reducing runtime significantly.

Since POLAR 3.0 runs much faster than existing detailed placement tools, detailed placement stage becomes the runtime bottleneck. We will try to parallelize existing detailed placement methods in our future work.

• Cell list	A, B, C, D, E, F, G, H, I, J, K, L
• Net list	red, blue, yellow
• Pin List	C, B, I, A, J, L, C, D, J
• Nets' first pin indexes	1, 4, 7
• Nets' last pin indexes	3, 6, 9
• region	(lx0, ly0, rx0, rx1)

• Cell ids	1(A), 2(B),3(C)
• Map ids	1->1, 2->2, 3->3
• Net ids	1(red), 2(blue), 3(yellow)
• Region	(lx1, ly1, rx1, ry1)

• Cell ids	7(G), 8(H), 9(I)
• Map ids	7->1, 8->2, 9->3
• Net ids	1(red)
• Region	(lx2, ly2, rx2, ry2)

• Cell ids	4(D), 5(E), 6(F)
• Map ids	4->1, 5->2, 6->3
• Net ids	3(yellow)
• Region	(lx3, ly3, rx3, ry3)

• Cell ids	10(J), 11(K), 12(L)
• Map ids	10->1, 11->2, 12->3
• Net ids	2(blue), 3(yellow)
• Region	(lx3, ly3, rx3, ry3)

Figure 4.7 Memory footprints for supporting partitioning efficiently. The placement instance is shown in Fig. 4.6.

CHAPTER 5. TPL LAYOUT GENERATION AND DECOMPOSITION WITH BLACK-BOX DETAILED PLACER

Triple patterning lithography (TPL) is considered one of the most promising solutions for sub-16nm technology nodes. To ensure that a layout after detailed placement is TPL decomposable, TPL constraints must be considered during detailed placement. Some previous works co-optimize detailed placement and layout decomposition by repeatedly performing layout decomposition during detailed placement steps. However, those algorithms are very time consuming. Besides, they only optimize wirelength, lithography conflict count and stitch count. Other placement objectives like timing and routability are hard to incorporate. In this chapter, we propose a new approach to perform detailed placement with TPL decomposition. Our approach is highly efficient and enables any existing detailed placer to be used as a black-box. The key idea is that we estimate how much space should be reserved around each type of standard cell such that a random placement is likely decomposable without any lithography conflict. By inflating each cell accordingly, any existing detailed placement tool can be used to generate a layout with the desired space reservation. This layout can then be easily refined to eliminate any lithography conflict at the cost of minimal cell displacement. Experimental results verify the effectiveness and efficiency of our approach.

5.1 Introduction

As the technology node scales to sub-16nm, double patterning lithography has reached its limits. TPL, which is a natural extension along the paradigm of multiple patterning lithography, is widely considered as one of the most promising solutions, along with extreme ultra violet, electron beam, and directed self-assembly.

In TPL, layout is decomposed into three masks. For any two layout features which are assigned to the same mask, a lithography conflict occurs if their distance is less than a threshold value d_{min}. To resolve lithography conflicts, stitches are used to split a feature into several parts, which could be assigned to different masks.

Many previous works on TPL layout decomposition assume that the layout is fixed and could not be modified. [146] proved that TPL layout decomposition is NP-complete, then proposed a semi-definite programming based approach. [147] pointed out that [146] would miss some legal stitches by the projection method [148], then proposed a better algorithm to find more legal stitches. [149] further improved the identification of legal stitches, then used graph simplification techniques to divide the problem into many smaller sub-problems. [150] tried to leverage existing DPL layout decomposition techniques with the help of an advanced data structure – SPQR tree. [151] considered TPL layout decomposition of row-structure layout. However, all these previous works cannot guarantee to generate solutions with minimum number of conflicts.

To ensure that a layout is decomposable, a pioneer work [152] suggested to consider lithography constraints during detailed placement. [153, 154] followed this suggestion, they considered DPL and TPL layout decomposition of row-structure layout respectively.

In [154], a branch and bound algorithm was applied to enumerate non-redundant coloring solutions for each type of standard cell in its first stage. Then in its second stage, TPL-aware detailed placement is performed to eliminate conflicts between standard cells. However, its second stage is time consuming and some complex placement quality metrics such as routability and timing are difficult to incorporate.

In this chapter, a simple and effective approach for TPL layout generation and decomposition is proposed with the help of any existing detailed placer. The key idea is that we estimate how much space should be reserved around each type of standard cell so that a random placement is likely decomposable without any lithography conflict. By inflating each cell accordingly, any existing detailed placer can be used as a black-box to generate a layout with the desired space reservation. This layout can be easily refined to eliminate any lithography conflict at the cost of minimal cell displacement. The contributions of our work are summarized as follows.

- A three-stage flow is proposed for TPL layout generation and decomposition. Our flow targets on not only satisfying TPL constraints, but also maintaining good placement quality by leveraging any existing detailed placer as a black-box.
- The desired space around each type of standard cell is effectively computed by a probabilistic method. By inflating the cells accordingly, any existing detailed placer can be used to generate a TPL-friendly and high-quality placement solution.
- A local cell swapping/recoloring and a displacement-driven cell packing are proposed to efficiently resolve lithography conflicts at the cost of minimal cell displacement.
- Experimental results show that our approach outperforms the state of the art, on both solution quality and CPU runtime.

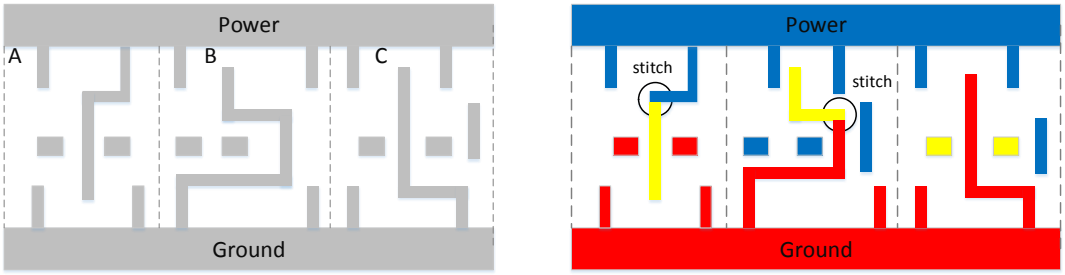
The rest of this chapter is organized as follows. Section 5.2 presents preliminaries. Section 5.3 is an overview of our approach. From Section 5.4 to Section 5.5, the details of our approach are illustrated. Experimental results are shown in Section 5.6. Finally, conclusions are made in Section 5.7.

5.2 Preliminaries

5.2.1 Problem definition

As in previous work [154], we assume that layout is a standard-cell based design with row structure. Given a standard cell library, the cells of different types are with the same height. Since the most complex layout features originate from metal 1 (M1) layer, in the rest of this chapter, we focus on M1 layer. In each row, the power/ground rails are going from the very left to the very right. Fig. 5.1 gives a simple row structure based layout.

The row structure layout has some good properties. [151, 154] pointed out that there was no lithography conflict between two M1 wires or contacts that are from different rows. Therefore, layout decomposition can be carried out for each row independently. Besides, the power/ground rails can be pre-colored without loss of generality. They can either be assigned the same color,



(a) Row structure layout (b) TPL decomposition

Figure 5.1 An instance of row structure layout and its TPL decomposition.

or different colors. The exact color to use is not an issue since we can generate other solutions from existing ones by permuting colors [151].

The problem can be formulated as follows. Given a standard cell library and an initial placement, the goal is to assign a coloring solution for each standard cell and perform detailed placement with respect to any placement metrics (e.g., wirelength, timing and routability) while minimizing the number of lithography conflicts and stitches.

5.3 Overview Of Our Approach

Basically, we divide our approach into three stages: (i) library pre-processing; (ii) black-box detailed placement; (iii) lightweight co-optimization. The overview of our approach is shown in Fig. 5.2.

In the first stage, standard cell pre-coloring is first performed. Then some look-up tables are constructed and the desired space around each type of standard cell is computed. All the information is stored in a database. Note that the first stage is only performed once off-line for all circuits using the same standard cell library.

In the second stage, standard cells are inflated according to their desired spaces from the database. And detailed placement is performed to generate a layout with desired space reservation. In this stage, any existing detailed placer can be leveraged as a black-box.

In the final stage, an initial layout decomposition (mask assignment) is performed based on the generated layout of the second stage. If all rows already have enough capacities (spaces)

to avoid lithography conflicts, we skip lightweight local cell swapping/recoloring. Otherwise, lightweight local cell swapping/recoloring is applied. At the end, displacement-driven cell packing is used to achieve TPL-friendly detailed placement at the cost of minimal cell displacement. Displacement is minimized in order to preserve the optimization performed in black-box detailed placement.

5.4 Library Pre-processing

5.4.1 Standard cell pre-coloring

In order to generate all non-redundant coloring solutions for each type of standard cell, we leverage the branch and bound algorithm used in [154]'s first stage. For i-th type of cell t_i, it has a set of coloring solutions denoted by $c_i^1, c_i^2, \cdots, c_i^{n_i}$, where n_i is the number of coloring solutions for t_i. The corresponding stitch counts for $c_i^1, c_i^2, \cdots, c_i^{n_i}$ are respectively $s_i^1, s_i^2, \cdots, s_i^{n_i}$. The width of t_i is denoted by w_i.

5.4.2 Construction of look-up tables

After all coloring solutions are generated, two look-up tables are built for future use.

The first one stores the minimal allowable spacing between two adjacent cells without resulting in conflict. For $d_{i,p}^{j,q}$, the left cell's type is t_i and its chosen coloring solution is c_i^p, and the right cell's type is t_j and its chosen coloring solution is c_j^q.

The second one $D\left((j,p),i,(k,q)\right)$ stores a pair (d,η) as shown in Formulas 5.1-5.2, where d is the minimal allowable spacing among three consecutive cells without resulting in conflict and η is the corresponding color solution of the middle cell. In here, the middle cell's type is t_i, its left neighbor's type is t_j and its chosen coloring solution is c_j^p, and its right neighbor's type is t_k and its chosen coloring solution is c_k^q.

$$D\left((j,p),i,(k,q)\right).d = \min_{1\le z\le n_i}\left\{d_{j,p}^{i,z} + d_{i,z}^{k,q}\right\} + w_i \tag{5.1}$$

$$D\left((j,p),i,(k,q)\right).\eta = \arg\min_{1\le z\le n_i}\left\{d_{j,p}^{i,z} + d_{i,z}^{k,q}\right\} \tag{5.2}$$

5.4.3 Standard cell desired space calculation

Lithography conflict between a pair of adjacent cells could be easily resolved as long as enough space is reserved in the vicinity of these two cells. By reversing desired space around each standard cell during detailed placement, a TPL-friendly layout is expected to generate. Since the final coloring solution for each standard cell is not determined yet, a probabilistic method is used to estimate the desired space.

For any type of standard cell t_i, suppose its left adjacent cell's type is t_j. Assuming that t_i and t_j use random coloring solution. To avoid lithography conflict, the expected minimal space $E_{i,j}$ between these two cells can be calculated by Formula 5.3. Then the desired space around t_i, denoted by μ_i, is defined by Formula 5.4. α is a parameter which is introduced to control the amount of space reserved. It is set to 0.9 for all the benchmarks used in our experiments. The Fig. 5.3 illustrates the idea. We will analyze the impact of α on solution quality in Section 5.7.

$$E_{i,j} = \frac{\sum_{1 \le p \le n_i} \sum_{1 \le q \le n_j} d_{i,p}^{j,q}}{n_i * n_j} \tag{5.3}$$

$$\mu_i = \alpha * \left(\frac{\sum_{1 \le j \le n} E_{j,i}}{2n} + \frac{\sum_{1 \le k \le n} E_{i,k}}{2n} \right) \tag{5.4}$$

5.5 Black-box Detailed Placement

The reserved space around each type of standard cell is provided to detailed placer as follows: for the standard cell whose type is t_i, its width is inflated to $w_i + \mu_i$. Then any existing detailed placer can be used as a black-box to generate a layout with desired space reservation. Besides, the placement metrics such as wirelength and timing, are taken care of by detailed placer (and this is the beauty of our approach). All cell widths are restored after this stage.

5.6 Lightweight Co-optimization

In this section, we focus on avoiding lithography conflicts. At the beginning, the sufficient condition of zero-conflict is pointed out. Then we present how to initialize mask (color) assignment based on the layout generated by black-box detailed placer. Next, coloring and placement are further refined by a lightweight local cell swapping/recoloring in order to satisfy the sufficient condition. Finally, it is displacement-driven cell packing.

5.6.1 Sufficient condition

The cell orderings of the generated layout by black-box detailed placer are preferred to maintain. We define threshold capacity for each row as follows.

Definition 8. *Threshold capacity is the minimal capacity that is enough to accommodate the cells without introducing lithography conflicts.*

It is easy to see that zero-conflict can be achieved if the threshold capacity of each row is less than its capacity. The threshold capacity of each row can be efficiently solved by the following dynamic programming.

The cells in the i-th row are denoted by $b_i^1, b_i^2, \cdots, b_i^{r_i}$ ordering from left to right, where r_i is the number of cells in the row. Their types are respectively $t_i^1, t_i^2, \cdots, t_i^{r_i}$. Let $K = \max_{1 \le i \le n} \{n_i\}$, where n is the number of standard cell types in library. K is the maximal number of coloring solutions over all standard cell types. A two-dimensional array $A\,[1..r_i]\,[1..K]$ is maintained, where $A[x][y]$ stores the leftmost position of cell b_i^x if it is assigned its y-th coloring solution. $A[x][y]$ can be computed recursively in Formula 5.5.

$$A[x][y] = \begin{cases} \infty & \text{if } y > n_{t_i^x} \\ 0 & \text{if } x = 1 \\ \min_{1 \le z \le n_{t_i^{x-1}}} \{A[x-1][z] + d^{t_i^x, y}_{t_i^{x-1}, z}\} + w_{t_i^{x-1}} & \text{otherwise} \end{cases} \tag{5.5}$$

The threshold capacity can be calculated by Formula 5.6.

$$\lambda_i = w_{t_i^{r_i}} + \min_{1 \le z \le n_{t_i^{r_i}}} A[r_i][z] \tag{5.6}$$

5.6.2 Initialize mask assignment

The threshold capacity is the absolutely minimal capacity required no matter how many stitches are used. However, to incorporate stitch count, in our dynamic programming routine, we use weighted leftmost position $A'[x][y]$ to replace $A[x][y]$ as follows, where $s^z_{t_i^x}$ is the stitch count and β is used to balance threshold capacity and stitch count.

$$A'[x][y] = \begin{cases} \infty & \text{if } y > n_{t_i^x} \\ 0 & \text{if } x = 1 \\ \min_{1 \le z \le n_{t_i^{x-1}}} \{A'[x-1][z] + d^{t_i^x, y}_{t_i^{x-1}, z} + \beta * s^z_{t_i^x}\} + w_{t_i^{x-1}} & \text{otherwise} \end{cases} \tag{5.7}$$

Given the placement by the black-box detailed placer, we apply the dynamic programming based on Formula 5.7 to obtain an initial mask assignment. Then we define request capacity as follows.

Definition 9. *Based on current coloring solution for each standard cell, request capacity is the minimal capacity that is enough to accommodate the cells without introducing lithography conflicts. For a row, if its request capacity is greater than its capacity, we call this row overflow, otherwise we call it underflow.*

If the number of overflow cells is 0, then we can move to displacement-driven cell packing. Otherwise, lightweight local cell swapping/recoloring is applied to eliminate all overflow rows.

5.6.3 Lightweight local cell swapping/recoloring

For any cell b_i^j (the j-th cell in i-th row) in an overflow row, we define a rectangular window covering b_i^j as its neighborhood. The cells within this neighborhood are called the neighbors of b_i^j. Suppose b_u^v (the v-th cell in the u-th row) is a neighbor of b_i^j, if either one of the two following cell swapping/recoloring conditions is satisfied, then they can be swapped and recolored.

1. The u-th Row is underflow. After swapping/recoloring, the request capacity λ_i of i-th row is decreased and the u-th row is still underflow.

2. The u-th row is overflow. After swapping/recoloring, the request capacity λ_i of i-th row and λ_u of u-th row are both decreased.

Let us look a case of local cell swapping/recoloring between b_i^j and b_u^v, where b_i^j and b_u^v have both left and right adjacent cells. The request capacity are calculated by Formula 5.8. We use η_i^j to denote the coloring solution of b_i^j. If swapping/recoloring is allowable, η_i^j and η_u^v are updated by Formula 5.8. For other cases, such as that b_i^j does not have left adjacent cell, the analysis is similar.

$$\begin{cases} \lambda_i & = \lambda_i^o - DST(\left(t_i^{j-1}, \eta_i^{j-1}\right), \left(t_i^j, \eta_i^j\right), \left(t_i^{j+1}, \eta_i^{j+1}\right)) + D(\left(t_i^{j-1}, \eta_i^{j-1}\right), t_u^v, \left(t_i^{j+1}, \eta_i^{j+1}\right)).d \\ \lambda_u & = \lambda_u^o - DST(\left(t_u^{v-1}, \eta_u^{v-1}\right), \left(t_u^v, \eta_u^v\right), \left(t_u^{v+1}, \eta_u^{v+1}\right)) + D(\left(t_u^{v-1}, \eta_u^{v-1}\right), t_{i,j}, \left(t_u^{v+1}, \eta_u^{v+1}\right)).d \end{cases} \tag{5.8}$$

$$\begin{cases} \eta_i^j & = D(\left(t_u^{v-1}, \eta_u^{v-1}\right), t_{i,j}, \left(t_u^{v+1}, \eta_u^{v+1}\right)).\eta \\ \eta_u^v & = D(\left(t_i^{j-1}, \eta_i^{j-1}\right), t_u^v, \left(t_i^{j+1}, \eta_i^{j+1}\right)).\eta \end{cases} \tag{5.9}$$

local cell swapping/recoloring may introduce stitches. To limit the use of stitches, a stitch budget is provided as input parameter. If the stitch budget is used up, the local cell swapping/recoloring that would lead to stitch increment is forbidden.

Local cell swapping/recoloring is presented in Algorithm 14. Lines 2-12 perform local cell swapping/recoloring. The neighbor cells of b_i^j are considered as swapping/recoloring candidates in first-come first-in order. Line 14 enlarges the neighborhood if the row is still overflow. The outer loop is never stopped until either i. the row is underflow or ii. the neighborhood is too big. According to our experimental results, the second criterion has never happened, because the layout generated by black-box detailed placer is already easy to decomposed. When it happens, it means that our approach could not eliminate all conflicts, but it still tries to optimize the total number of conflicts. To limit cell displacement, the initial neighborhood for cell b_i^j is set

to $x \in [x_i^j - 3 * w_{t_i^j}, x_i^j + 3 * w_{t_i^j}], y \in [y_i^j - H, y_i^j + H]$, where H is the standard row height, (x_i^j, y_i^j) is the coordinate of b_i^j. Each time the neighborhood is enlarged by $2 * w_{t_i^j}$ in x-direction and $2 * H$ in y-direction.

Algorithm 14 Local cell swapping/recoloring to eliminate overflow row

Require: Placement is legalized. The i-th row is overflow. Stitch budget S.
Ensure: The i-th row is underflow.

```
1: while the i-th row is overflow do
2:     for each pair of (b_i^j, b_u^v) do
3:         if b_i^j and b_u^v can be swapped then
4:             Swap b_i^j and b_u^v;
5:             Update η_i^j and η_u^v;
6:             Update λ_i and λ_u;
7:             Update S;
8:             if S < 0 then
9:                 Cancel swap/recolor and recover S;
10:            end if
11:        end if
12:    end for
13:    if the i-th row is overflow then
14:        Enlarge the neighborhood;
15:        if neighbourhood is too big then
16:            Cannot obtain zero-conflict solution and exit;
17:        end if
18:    end if
19: end while
```

Algorithm 15 always either return a layout which could be decomposed without introducing conflicts or exit in line 16 based on the following observation. According to cell swapping/recoloring conditions, if case (i) happens, then λ_i is reduced without introducing additional overflow row. If case (ii) happens, both λ_i and λ_u are reduced. Therefore, in both cases, λ_i is reduced without introducing additional overflow rows. So the theorem is established.

5.6.4 Displacement-driven cell packing

The goal of cell packing is to minimize cell displacement with respect to the placement by black-box detailed placer. We define displacement-driven cell packing problem as follows.

Definition 10. *Displacement-driven cell packing: Given a detailed placement, find positions and coloring solutions for cells in each row under constraint that cell orderings are fixated. The objective is to minimize cell displacement, conflict and stitch count.*

Displacement-driven cell packing can be solved optimally by simply modifying the TPL-OSR's optimal algorithm proposed by [154]. However, its time complexity is $O(Kr_i m_i)$, where m_i and r_i are respectively the number of sites and cells in the i-th row. So it is very expensive.

We propose a two-step speed-up heuristic method only sacrificing insignificant quality. In the first step, we assign coloring solution to each cell by the dynamic programing approach based on Formula 5.7. In the second step, only cell shifting is allowed to optimize the displacement. In [103], an efficient single-segment clustering algorithm was proposed to solve fixed-order single segment placement problem optimally. In our method, the second step has similarity with fixed-order single segment placement problem. There are two differences. Firstly, the objective is displacement rather than wirelength. Since cell displacement of cell b_i^j can be modeled easily by adding a two-pin pseudo net from a fixed pseudo pin located in the current position of b_i^j to b_i^j, the wirelengh of this pseudo net is equal to the displacement of b_i^j. Secondly, the minimal distance between two adjacent cells without introducing conflicts should be enforced. This can be easily realized by cell inflation. Therefore, [103]'s single-segment clustering algorithm can be reused.

The time complexity of our two-step heuristic method is $O(r_i K)$. The first step needs $O(r_i K)$, while the second step needs $O(r_i)$ [103]. Compared with TPL-OSR's optimal algorithm [154], our method is faster by $O(m_i)$.

For a row, if the above method could not achieve zero-conflict, we return to TPL-OSR's optimal algorithm [154]. Based on our experimental results, for all the benchmarks, our two-step heuristic method works very well and it is extremely fast.

5.7 Experimental Results

5.7.1 Experimental configuration

Our approach is implemented in C++, and the latest version of [154]'s C++ implementation is also obtained. Both are run on the same Linux machine with 3.4GHz CPU, 4GB main memory. Benchmarks are the same as [154]'s. FastDP [103] is used as black-box detailed placer in our approach.

Table 5.1 The characteristics of benchmarks used in TPL-aware detailed placement

benchmark	$\|V\|$	$\|E\|$	Util(%)	infl (%)
alu-70	2110	2445	70.00	4.55
alu-80	2110	2445	80.22	5.53
alu-90	2119	2445	90.14	5.99
byp-70	4416	5732	70.01	2.64
byp-80	4416	5732	80.04	3.04
byp-90	4416	5732	90.14	3.53
div-70	3758	4487	70.14	3.65
div-80	3758	4487	80.14	4.16
div-90	3758	4487	90.12	4.63
ecc-70	1322	1552	70.13	2.82
ecc-80	1322	1552	80.10	3.18
ecc-90	1322	1552	90.25	3.72
efc-70	1183	1314	70.03	2.77
efc-80	1183	1314	80.26	3.39
efc-90	1183	1314	90.27	3.79
ctl-70	1694	2039	70.04	2.73
ctl-80	1694	2039	80.00	3.06
ctl-90	1694	2039	90.24	3.70
top-70	14793	15448	70.08	4.00
top-80	14793	15448	85.10	4.58
top-90	14793	15448	90.06	5.10

The characteristics of benchmarks are presented in Table 5.1. The second column $|V|$ is the number of movable cells. The third column $|E|$ is the number of nets. The design utilization is in the forth column and cell inflation ratio (the ratio of extra inflated area and die area) is given in the last column. It shows that by inflating cells just a little bit, a TPL-friendly layout can be generated by existing detailed placers.

5.7.2 Space reservation analysis

We have studied the effect of α in Formula 5.4 to the results. If α is too large, too much space would be reserved and the placement quality would be degraded. If α is too small, there may not be enough space to resolve all lithography conflicts. In our experimental results, we notice that as long as α is between 0.6 and 1.1, the results are good and are not very sensitive to the value of α. In our experiments, we set α to 0.9 for all benchmarks. It is not surprising that the value of α can be less than 1. It is because the cell coloring solutions can be optimized to make the space required to be less than the average.

5.7.3 Runtime analysis

The look-up tables are only built once off-line and stored in the main memory to be reused. (The memory used to store look-up table is less than 100M bytes). Therefore, the CPU runtime of building look-up table are excluded for both [154] and our approach. The CPU runtime breakdown of our approach is shown in Fig. 5.4, where the y-axis is the CPU runtime percentage. We can see that calling the black-box detailed placer dominates the total CPU runtime, so our approach has similar scalability with the black-box detailed placer and is very suitable for large scale design.

5.7.4 Solution quality comparison with previous work

The experimental results are listed in Table 5.2. Compared with [154]'s slow mode, on average, our approach achieves 4.2% improvement on wirelength, 33% reduction on the number of stitches and about 96.9× speed up on CPU runtime. Compared with [154]'s fast mode (which reports error on alu_90), on average, our approach achieves 6.5% improvement on wirelength,

4% reduction on the number of stitches and about 2.66× speedup. Besides, it can be observed that our approach is very suitable for the circuits which have high utilization, such as alu_90, div_90 and efc_90. While [154] does not perform well on these circuits.

Table 5.2 Experimental results of TPL-aware detailed placement

benchmark	[154]'s slow mode				[154]'s fast mode				our approach			
	CN	ST	WL	runtime(s)	CN	ST	WL	runtime(s)	CN	ST	WL	runtime(s)
alu-70	0	880	2.622	28.8	0	622	2.637	0.57	0	610	2.563	0.14
alu-80	0	926	2.613	26.7	0	624	2.657	0.69	0	610	2.561	0.16
alu-90	0	1054	2.965	34.5	0	*	*	*	0	610	2.566	0.41
byp-70	0	1375	11.78	47.9	0	1176	11.78	0.94	0	1134	11.52	0.41
byp-80	0	1544	11.51	46.9	0	1180	11.53	1.79	0	1134	11.28	0.36
byp-90	0	1781	11.85	91.2	0	1175	12.16	1.75	0	1134	11.49	0.96
div-70	0	1555	4.793	23.3	0	1384	4.799	0.64	0	1316	4.668	0.45
div-80	0	1713	4.602	43.5	0	1375	4.616	1.22	0	1316	4.506	0.39
div-90	0	1838	4.691	71.5	0	1371	5.380	1.75	0	1316	4.442	0.76
ecc-70	0	278	2.071	6.31	0	262	2.071	0.15	0	258	2.029	0.15
ecc-80	0	290	1.955	5.72	0	260	1.960	0.26	0	258	1.928	0.25
ecc-90	0	325	1.969	12.5	0	259	1.982	0.24	0	258	1.872	0.3
efc-70	0	748	1.054	5.18	0	688	1.056	0.14	0	671	1.000	0.2
efc-80	0	823	1.015	4.74	0	691	1.022	0.12	0	671	0.987	0.2
efc-90	0	885	1.164	11.1	0	695	1.377	0.32	0	671	1.015	0.35
ctl-70	0	333	2.529	8.45	0	297	2.530	0.22	0	275	2.473	0.24
ctl-80	0	381	2.376	8.08	0	293	2.385	0.2	0	275	2.346	0.35
ctl-90	0	441	2.388	15.8	0	287	2.418	0.55	0	275	2.335	0.41
top-70	0	5548	17.43	204	0	4803	17.45	4.59	0	4731	16.68	1.36
top-80	0	6124	16.49	193	0	4832	16.53	9.09	0	4731	15.95	1.14
top-90	0	6822	16.78	430	0	4789	19.72	13.1	0	4731	15.81	1.15
norm	1	1.33	1.042	96.9	1	1.04	1.065	2.66	1	1	1	1

5.8 Conclusions

In this chapter, we propose an efficient approach for TPL layout generation and decomposition. The key idea is that by space reservation and cell inflation, a layout generated by detailed placement is already TPL-friendly. And the lithography conflicts can be easily eliminated by lightweight method. Besides, by leveraging existing detailed placers, unlike [154], we do not need to develop new algorithms to incorporate different placement metrics (e.g, timing and routability). Our approach maximally maintains the quality of a given initial placement, and its scalability is also determined by the black-box detailed placer used.

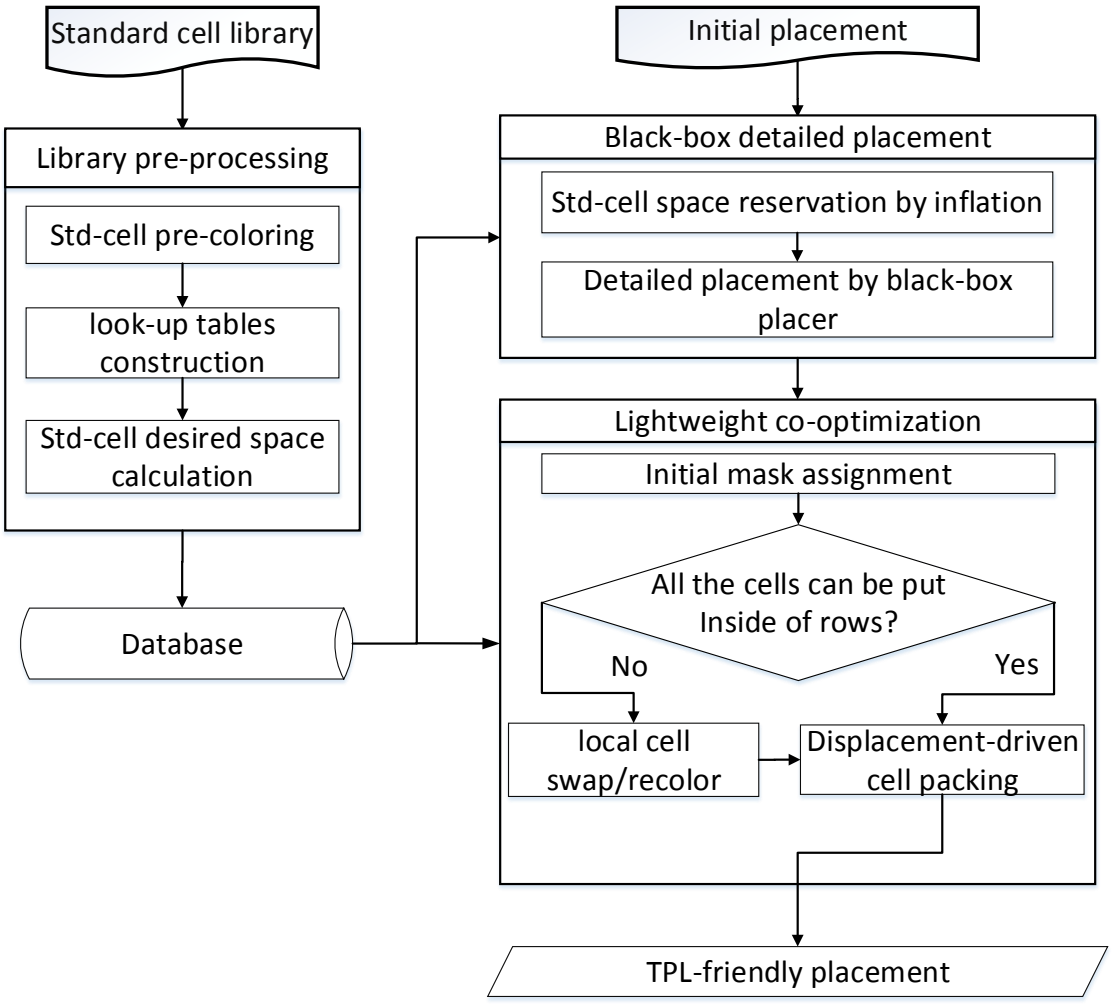


Figure 5.2 The overview of our TPL-aware detailed placement approach.

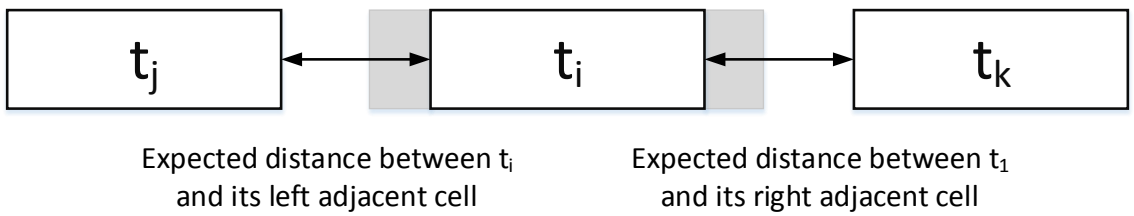


Figure 5.3 Space reservation: the shadow parts are reserved space.

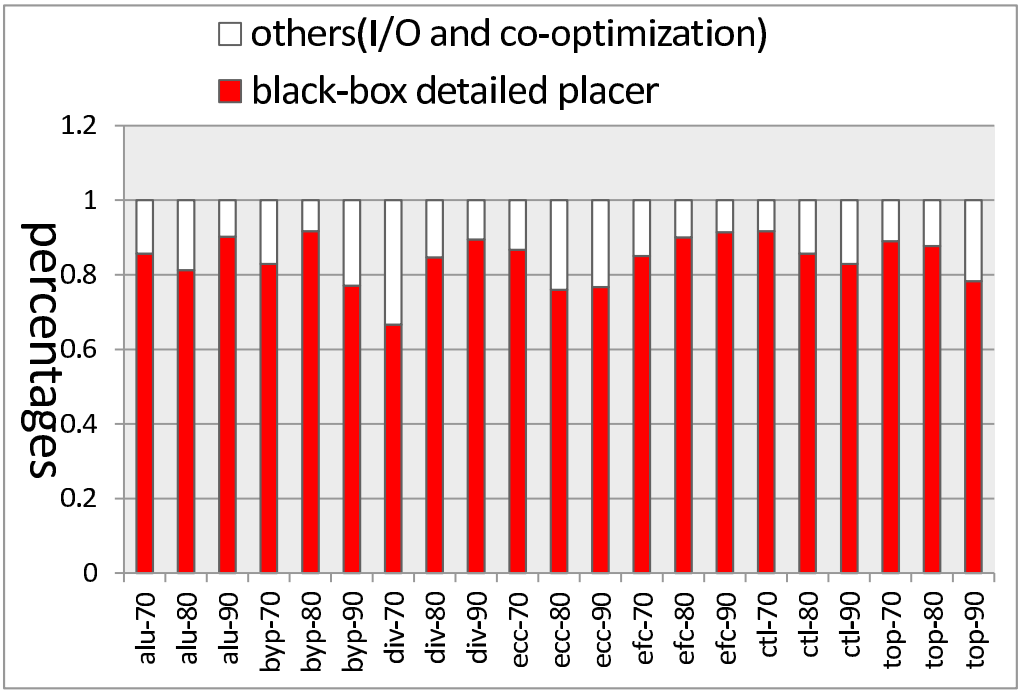


Figure 5.4 The runtime breakdown of our TPL-aware detailed placement approach.

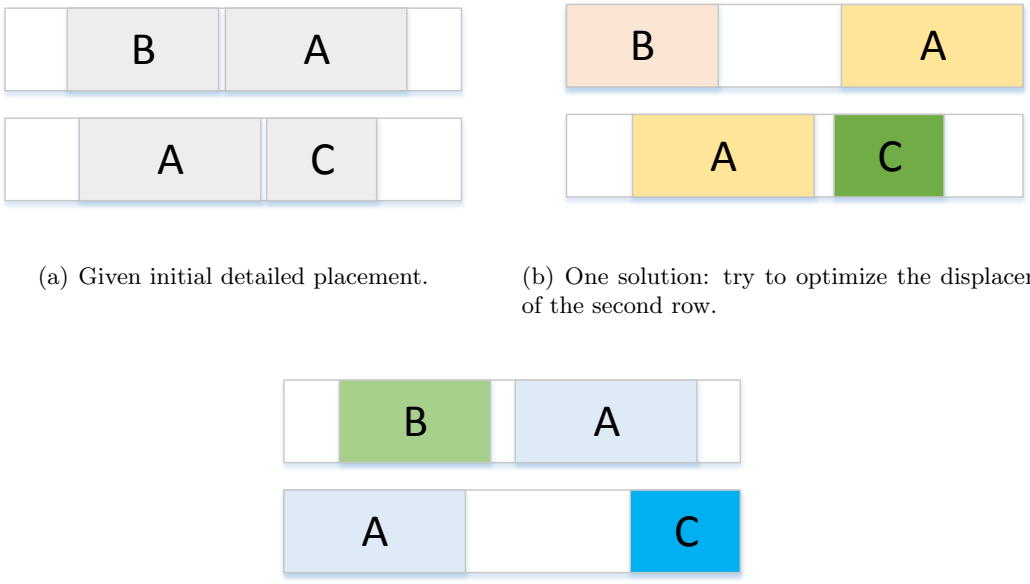
CHAPTER 6. TPL-AWARE DETAILED PLACEMENT REFINEMENT WITH COLORING CONSTRAINTS

To minimize the effect of process variation for a design in triple patterning lithography (TPL), it is beneficial for all standard cells of the same type to share a single coloring solution. In this chapter, we investigate the TPL-aware detailed placement refinement problem under these coloring constraints. Given an initial detailed placement, the positions of standard cells are perturbed and a TPL solution complying with the coloring constraints is derived while minimizing cell displacement, lithography conflicts and stitches. We prove that this problem is NP-complete and show that it can be formulated as a mixed integer linear program. Since mixed integer linear programming is very time consuming, we propose an effective heuristic algorithm. In our approach, important adjacent pairs of standard cells are recognized firstly, since they have significant impact on cell displacement. Then a tree-based heuristic is applied to generate a good initial solution for our linear programming-based refinement. Experimental results show that compared with mixed integer linear programming, our heuristic approach is comparable in solution quality while using very short CPU runtime.

6.1 Introduction

With the technology node scaling to sub-16nm, electron beam (E-beam), extreme ultraviolet lithography (EUVL) and TPL are considered the most promising lithography technologies. In this chapter, we are focusing on TPL.

There are many previous works on TPL optimization. The fundamental problem of TPL is to eliminate lithography conflicts while minimizing stitch count. [146, 147, 149, 150, 151, 155, 156, 157] are related to TPL layout decomposition. [146, 147, 149, 150] focus on 2-Dimension



(a) Given initial detailed placement.

(b) One solution: try to optimize the displacement of the second row.

(c) Another solution: try to optimize the displacement of the first row.

Figure 6.1 An instance of problem: choosing different coloring solutions for types A, B and C plus cell shifting.

layout decomposition. [151, 155] focus on row-based 1-Dimension layout decomposition. [158, 159] consider TPL during detailed routing stage.

Recently, [154] presents a TPL aware detailed placement approach in which layout decomposition and placement are resolved simultaneously. The approach is effective in resolving lithography conflicts. However, the approach only considers the optimization of wirelength together with lithography conflicts and stitch number. It is not clear how to incorporate other placement objectives like timing and routability.

Besides, [155] points out the advantage of assigning the same lithography pattern for the same standard cell type during TPL layout decomposition. This would minimize the effect of process variation and best guarantee that those standard cells of the same type eventually have similar physical and electrical characteristics. However, [155] only considers the decomposition of a fixed layout, and hence often cannot completely satisfy these constraints.

In this chapter, we investigate the TPL-aware detailed placement refinement problem under the coloring constraints that all standard cells of the same type should share the same TPL coloring solution. Given an initial detailed placement, the positions of standard cells are perturbed and a TPL solution complying with the coloring constraints is derived while minimizing total cell displacement, lithography conflicts and stitches simultaneously.

Different from [154], our approach is applied to an optimized detailed placement under any conventional placement metrics. By refining it with minimal perturbation, the quality of the detailed placement can be preserved. In addition, we consider the coloring constraints. Compared with [155], as placement perturbation is allowed, the coloring constraints are always satisfied in our approach. We prove that this problem is NP-complete and show that it can be formulated as a mixed integer linear program (MILP). Since the MILP is time consuming to solve, we propose an effective heuristic algorithm to solve it. In our algorithm, important adjacent pairs of standard cells are recognized firstly, since they have significant impact on cell displacement. Then a tree-based heuristic is applied to generate a good initial solution which is then refined by a linear programming (LP)-based technique. Experimental results show that compared with MILP solution, the heuristic method is comparable in solution quality while using very limited CPU runtime. The contributions of this chapter are summarized as follows.

- We formulate a new TPL optimization problem considering TPL coloring constraints for standard cells during detailed placement.
- We prove that this new problem is NP-complete.
- We propose a MILP formulation for this new problem.
- Since MILP is very time consuming to solve, we propose an effective heuristic algorithm.

The rest of this chapter is organized as follows. In Section 6.2, we gives the formal problem definition and its MILP formulation. In Section 6.3, we prove that this problem is NP-complete. In Section 6.4, we illustrate the heuristic algorithm. In Section 6.5, we present the experimental results. Finally, we make our conclusions in Section 6.6.

6.2 Problem Definition

Given a standard cell library, all feasible coloring solutions for each cell type are found out firstly. Since each cell contains only a small number of layout features, the enumerative approach proposed in [154] works well. Besides, this step is performed once per library. For the i-th type of cell denoted by t_i, there are n_i feasible coloring solutions $p_i^1, p_i^2, \cdots, p_i^{n_i}$. The corresponding stitch counts are $s_i^1, s_i^2, \ldots, s_i^{n_i}$. The width of t_i is w_i. There are k types of standard cells in the library. Given a detailed placement, which has n rows. For the j-th row, the types of standard cells ordered from left to right are $c_j^1, c_j^2, \cdots, c_j^{r_j}$, where r_j is the number of cells in the j-th row.

The TPL-aware displacement-driven detailed placement with coloring constraints is defined as follows.

Given a standard cell library with a set of feasible coloring solutions for each standard cell type, and an initial detailed placement, eliminate all lithography conflicts by choosing one coloring solution for each type of standard cell and shifting the standard cells without changing the cell ordering in each row. The objective is to minimize the total cell displacement and the number of stitches.

Fig. 6.1 gives an instance of this problem. By choosing coloring solutions for types A, B and C and shifting cells, conflicts are eliminated. In Fig. 6.1(a), an initial detailed placement with two rows is given. In Fig. 6.1(b), cell displacement of the second row is optimized well while that of the first row is not. On the contrary, in Fig. 6.1(c), cell displacement of the first row is optimized well while that of the second row is not. It shows that different TPL solutions may lead to significantly different cell distribution in each row.

6.2.1 MILP formulation

The above problem can be formulated as a MILP. We use a binary variable b_i^j to denote whether the coloring solution p_i^j is assigned to standard cell type t_i. In the i-th row, the original central x-coordinates of cells ordered from left to right are $o_i^1, o_i^2, \cdots, o_i^{r_i}$, their new central x-coordinates are $x_i^1, x_i^2, \cdots, x_i^{r_i}$, their displacement are $q_i^1, q_i^2, \cdots, q_i^{r_i}$. For any two adjacent

cells, the type of left one is t_i and its coloring solution is p_i^u, the type of right one is t_j and its coloring solution is p_j^v. To avoid lithography conflict, the minimal distance between these two cells is a constant denoted by $d_{i,j}^{u,v}$. For any two adjacent cells in the row i, let x_i^{j-1} and x_i^j be their central x-coordiniates, their actual distance is denoted by z_i^j. Besides, the width W of placement region is also given. The problem can be formulated into the following mathematical programming. Note that in this chapter, for any pair of adjacent cells, the distance is from the center of the left one to the center of the right one.

$$\text{Minimize: } \alpha \sum_{i=1}^{n} \sum_{j=1}^{r_i} \sum_{k=1}^{n_{c_i^j}} b_{c_i^j}^k \times s_{c_i^j}^k + \beta \sum_{i=1}^{n} \sum_{j=1}^{r_i} q_i^j$$

S.T:

$$\sum_{j=1}^{n_i} b_i^j = 1, \qquad \forall 1 \le i \le k$$

$$x_i^j - x_i^{j-1} = z_i^j, \qquad \forall 1 \le i \le n \wedge 2 \le j \le r_i$$

$$z_i^j \ge \sum_{u=1}^{n_{c_i^{j-1}}} \sum_{v=1}^{n_{c_i^j}} b_{c_i^{j-1}}^u \times b_{c_i^j}^v \times d_{c_i^{j-1},c_i^j}^{u,v}, \quad \forall 1 \le i \le n \wedge 2 \le j \le r_i$$

$$x_i^j - o_i^j \le q_i^j, \qquad \forall 1 \le i \le n \wedge 1 \le j \le r_i$$

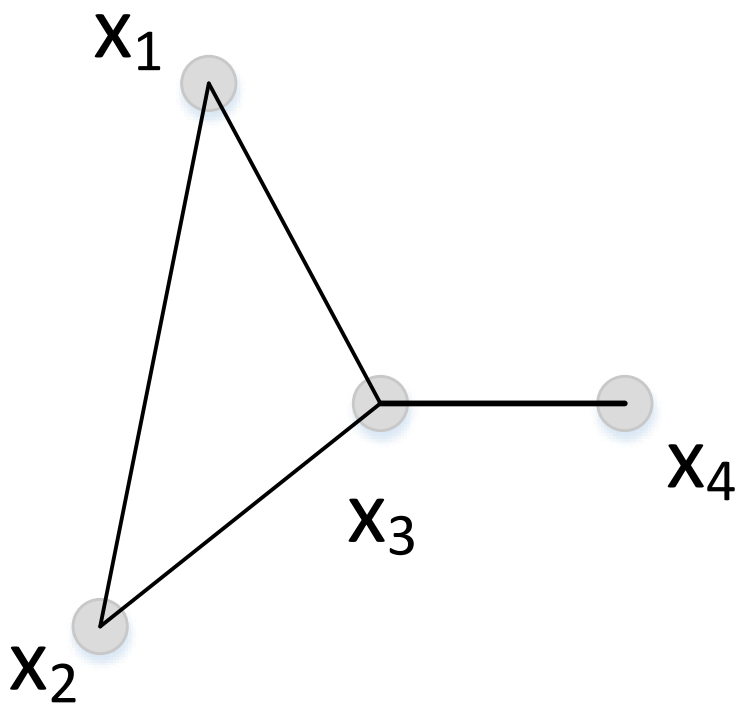
$$o_i^j - x_i^j \le q_i^j, \qquad \forall 1 \le i \le n \wedge 1 \le j \le r_i$$

$$x_i^j \ge \frac{w_{c_i^j}}{2}, \qquad \forall 1 \le i \le n \wedge 1 \le j \le r_i$$

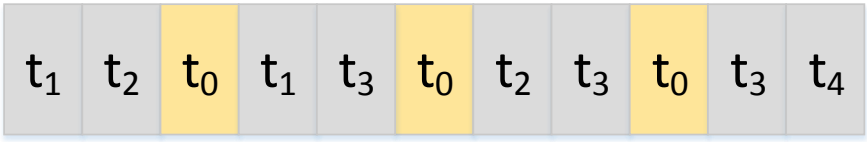
$$x_i^j \le W - \frac{w_{c_i^j}}{2}, \qquad \forall 1 \le i \le n \wedge 1 \le j \le r_i$$

$$b_i^j = 0 \quad or \quad 1, \qquad \forall 1 \le i \le k \wedge 1 \le j \le n_i$$

The objective is a weighted sum of total cell displacement and stitch count. The first constraint represents that standard cells of the same type should have the same coloring solution. The second and third constraints represent that for any two adjacent cells, there is enough distance to avoid lithography conflict. The fourth and fifth constraints represent cell displacement. Finally, the last two constraints mean that cells should be put inside of placement region. The product of two binary variables in the third constraint can be transformed into linear constraints as follows: $c = a * b \Leftrightarrow a + b - c \le 1 \wedge a - c \ge 0 \wedge b - c \ge 0$, where a, b, c are all binary variables. Therefore, the problem can be formulated as a MILP.



(a) An instance of 3-coloring problem. The three colors are RED, BLUE and GREEN.



(b) An instance of single-row version. The widths of cells are 1. The width of row is 11. For any type of standard cell t_i, it has three feasible coloring solutions $\left(p_i^1, p_i^2, p_i^3\right)$. p_i^1, p_i^2 and p_i^3 are respectively corresponding to RED, BLUE and GREEN.

Figure 6.2 The reduction from 3-coloring problem to single-row version.

6.3 Complexity Of Problem

To see the complexity of this problem, let us look at a special version of its decision problem firstly.

Definition 11 (Single-row version)**.** *The given initial detailed placement has only one row. The problem is to decide whether there is a feasible solution to accommodate all cells without conflicts.*

Theorem 1. *The single-row version is NP-complete.*

Proof. It is easy to see that the single-row version is NP. We show that the 3-coloring problem can be reduced to single-row version. Since the 3-coloring is NP-complete [160], the single-row version is NP-complete.

Suppose in a 3-coloring problem instance, there are n nodes denoted by $x_1, x_2, \cdots, x_n$. There are m edges denoted by $e_1, e_2, \cdots, e_m$. We can construct the following single-row version instance.

Each node x_i is corresponding to one type of standard cell t_i, which has three feasible coloring solutions p_i^1, p_i^2, p_i^3. p_i^1, p_i^2 and p_i^3 are corresponding to RED, BLUE and GREEN respectively. There is a special type of standard cell t_0. The width of standard cells are all 1.

We define the minimal distance between t_i and t_j to eliminate conflict as follows.

$$d_{i,j}^{u,v} = \begin{cases} 1 & \text{if } u \neq v \text{ and } i \neq 0 \text{ and } j \neq 0 \\ 2 & \text{if } u = v \text{ and } i \neq 0 \text{ and } j \neq 0 \\ 1 & \text{if } i = 0 \text{ or } j = 0 \end{cases}$$

It means that for any pair of adjacent cells, if the type of either one is t_0, the minimal distance between these two cells to avoid conflict is 1 no matter what the final coloring solutions are. Otherwise, if the left one is assigned the coloring solution which is corresponding to p_i^k $(1 \leq k \leq 3)$ and the right one is assigned the coloring solution which is corresponding to p_j^k,

the minimal distance between these two cells to avoid lithography conflict is 2. Otherwise the minimal distance is 1.

For any two nodes x_i and x_j, suppose $i < j$ without loss of generality. If there is an edge $e = (x_i, x_j)$, then we construct a pair of adjacent cells (t_i, t_j). Besides, we add a standard cell of type t_0 between any two pairs of constructed adjacent cells. And the width of row is defined as the number of constructed standard cells, i.e., 3m-1. Fig. 6.2(b) shows the corresponding single-row version instance of the 3-coloring problem instance in Fig. 6.2(a).

If the above 3-coloring problem instance is true, then in the constructed single-row version instance, for any two adjacent cells t_i and t_j $(i < j)$, we can choose the coloring solutions so that the minimal distance between these two cells to avoid lithography conflict is 1. Therefore, all the constructed standard cells can be put inside of the row. Similarly, if single-row version instance is true, then we can find a solution that satisfies the corresponding 3-coloring problem instance.

□

The displacement-driven TPL-aware detailed placement with ordering and coloring constraints is a generalization of the single-row version, so it is also NP-complete [160].

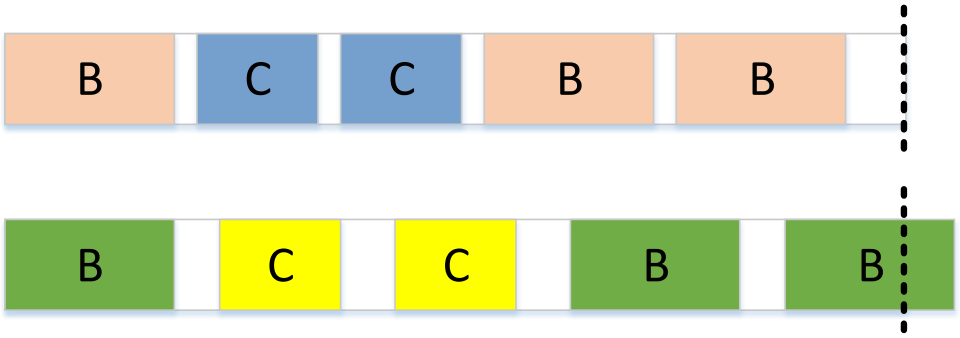
6.4 Methodology

Since the problem is NP-complete and MILP is very time consuming, we propose an effective heuristic algorithm to solve this problem. In this section, we firstly show the motivation of our approach. Next, we present its overview which is composed of three stages. Finally, we illustrate these three stages respectively.

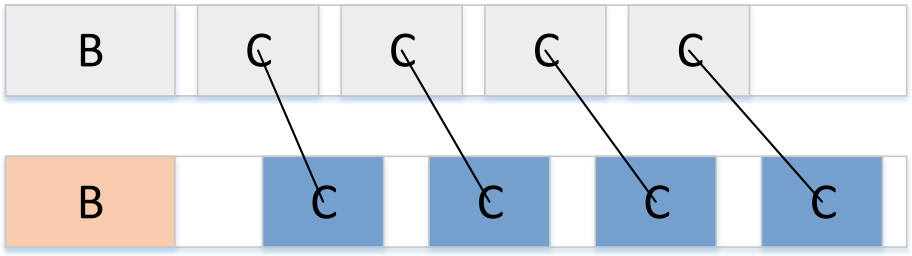
6.4.1 Motivation

Since standard cells of the same type should have the same coloring solution, we define adjacent pair as follows.

Definition 12. *An adjacent pair is a pair of types of two adjacent standard cells.*



(a) The upper figure represents that all the cells are put inside of row if the minimal distances (to eliminate lithography conflicts) of adjacent pairs are optimized well. On the contrary, in the lower figure, the right most cell B is outside of row if those minimal distances of adjacent pairs are not optimized well.



(b) In the upper figure which represents the original placement, the left-most adjacent pair (cell B and cell C) is the most important one to optimize cell displacement. If the minimal distance of this pair to eliminate conflict is not optimized well, all the other cells on the right hand side would be shifted right as shown in lower figure.

Figure 6.3 The two examples reveal the motivation of our heuristic approach.

For example, if the type of left cell is t_i and the type of right one is t_j, the corresponding adjacent pair is (t_i, t_j). The minimal distances of adjacent pairs to avoid lithography conflicts have significant impact on solution quality of this problem. There are two reasons. Firstly, if these minimal distances are not optimized well, then it would be difficult to put all cells inside of the row region, as shown in Fig. 6.3(a). Secondly, different adjacent pairs have different impact on total cell displacement, as shown in Fig. 6.3(b). Therefore, our method tries to focus on the minimal distances of important adjacent pairs.

6.4.2 Overview

Our approach is composed of three stages. In the first stage, we propose a method to recognize the important adjacent pairs. In the second stage, we try to optimize minimal distances of important adjacent pairs and a tree-based heuristic is applied to get a good initial solution. In the last stage, we apply LP-based method to refine the solution. The overview is presented in Fig. 6.4.

6.4.3 Important adjacent pair recognition

We use a positive integer to represent how important an adjacent pair is. We call this integer the weight of adjacent pair. Higher weight means more important. For example, as shown in Fig. 6.3(b), apparently, the adjacent pair (B, C) should have the highest weight. We use $weight[i][j]$ to denote the weight of adjacent pair (t_i, t_j).

At this stage, we do not know what the final coloring is. Therefore, we propose a simple method to estimate the new cell distribution. For any adjacent pair (t_i, t_j), we calculate the average minimal distance $d_{i,j}^{ave}$ to avoid lithography conflict. This value is given by the following Formula 6.1.

$$d_{i,j}^{ave} = \frac{\sum_{u=1}^{n_i} \sum_{v=1}^{n_j} d_{i,j}^{u,v}}{n_i * n_j} \tag{6.1}$$

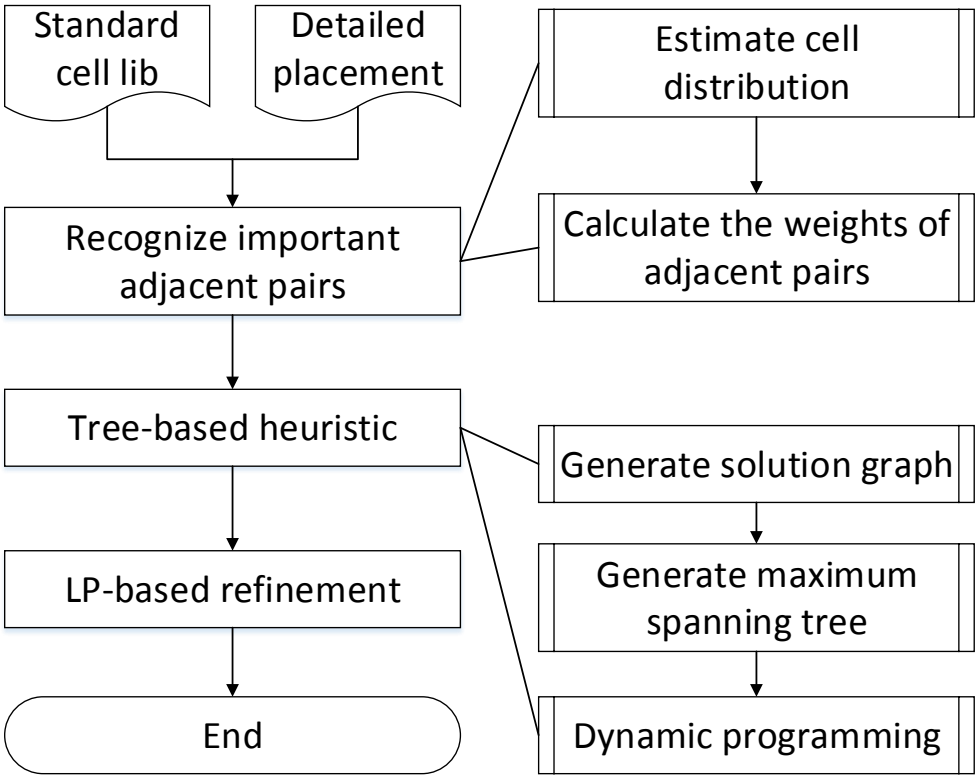


Figure 6.4 The overview of our heuristic approach.

The minimal total cell displacement can be achieved by LP as follows.

Minimize: $\sum_{i=1}^{n} \sum_{j=1}^{r_i} q_i^j$

Subject to:

$$x_i^j - x_i^{j-1} \geq d^{ave}_{c_i^{j-1}, c_i^j}, \forall 1 \leq i \leq n \wedge 2 \leq j \leq r_i$$

$$x_i^j - o_i^j \leq q_i^j, \forall 1 \leq i \leq n \wedge 1 \leq j \leq r_i$$

$$o_i^j - x_i^j \geq q_i^j, \forall 1 \leq i \leq n \wedge 1 \leq j \leq r_i$$

$$x_i^j \geq \frac{w_{c_i^j}}{2}, \forall 1 \leq i \leq n \wedge 1 \leq j \leq r_i$$

$$x_i^j \leq W - \frac{w_{c_i^j}}{2}, \forall 1 \leq i \leq n \wedge 1 \leq j \leq r_i$$

Then we define shifting direction of standard cell below.

Definition 13. *For the j-th standard cell in row r_i, its shifting direction is left if $x_i^j < o_i^j$, and right if $x_i^j > o_i^j$, otherwise no shifting. We use $\rightarrow$ to denote left shifting, $\leftarrow$ for right shifting, and $=$ for no shifting.*

Algorithm 16 gives the method to calculate the weights of adjacent pairs. The idea is that for a pair of adjacent cells, if their minimal distance to eliminate conflict is increased, the weight of this pair would roughly reflect the increment of total cell displacement. Let us look at an example. A placement row contains six cells and five adjacent pairs. The shifting directions of these six cells are $\rightarrow, \rightarrow, \rightarrow, \rightarrow, \rightarrow, \rightarrow$. The five adjacent pairs' weights ordered from left to right are respectively 5, 4, 3, 2 and 1. The weight of the left-most one is 5, because if its minimal distance is increased by 1 unit, the total cell displacement would be increased by 5 units roughly.

Algorithm 15 Method to calculate the weights of adjacent pairs

```
1: Calculate d_{i,j}^{ave} for each pair of adjacent pair (t_i, t_j);
2: Solve the LP to get the shifting direction of each standard cell;
3: for each placement row do
4:     [start, end] is the index range of cells (in ascending order of their x-coordinate) in this row;
5:     for any adjacent pair P = (t_i, t_j) in the row do
6:         ll and rr are the indexes of t_i and t_j in the row;
7:         if the left cell is ← then
8:             for k from ll to start do
9:                 if the cell whose order is k is ← or '=' then
10:                    weight[i][j]+ = 1;
11:                else
12:                    break;
13:                end if
14:            end for
15:        end if
16:        if the right cell is → then
17:            for k from ll + 1 to end do
18:                if the cell whose order is k is → or '=' then
19:                    weight[i][j]+ = 1;
20:                else
21:                    break;
22:                end if
23:            end for
24:        end if
25:    end for
26: end for
```

6.4.4 Tree-based heuristic

After the weights of all adjacent pairs are computed, a solution graph can be constructed as follows. In the solution graph, each node represents a standard cell type. The edge between two nodes represents an adjacent pair.

Let f_i be the coloring solution that standard cell type t_i uses. The cost $cost_i$ of node t_i and the cost $cost_{i,j}$ of edge connecting t_i and t_j in the solution graph are defined in the following two Formulas 6.2 and 6.3.

$$cost_i[f_i] = \beta * weight[i][i] * d_{i,i}^{f_i,f_i} + \alpha * s_i^{f_i} \tag{6.2}$$

$$cost_{i,j}[f_i, f_j] = \beta * [weight[i][j] * d_{i,j}^{f_i,f_j} + weight[j][i] * d_{j,i}^{f_j,f_i}] \tag{6.3}$$

The purpose of our tree-based heuristic is to find the coloring solution for each standard cell type, so that the total cost including cost of nodes and edges in the solution graph is minimized. It is not hard to see that if solution graph is of a tree structure, then dynamic programming can be applied to get the optimal coloring solution. Fortunately, it is observed that solution graphs for industrial benchmarks are sparse graphs. Next, we propose a method to leverage this observation.

6.4.4.1 Maximum spanning tree generation

The basic idea to leverage the observation is to ignore some relatively less important adjacent pairs and turn the solution graph into a tree. The cost of each edge connecting t_i and t_j in solution graph is replaced by Formula 6.4, where $d_{i,j}^{max}$ and $d_{i,j}^{min}$ are defined in Formula 6.5 and 6.6 respectively.

$$cost'_{i,j} = \alpha * [weight[i][j] * \left(d_{i,j}^{max} - d_{i,j}^{min}\right) + weight[j][i] * \left(d_{j,i}^{max} - d_{j,i}^{min}\right)] \tag{6.4}$$

$$d_{i,j}^{max} = \max_{1 \le u \le n_i} \max_{1 \le v \le n_j} d_{i,j}^{u,v} \tag{6.5}$$

$$d_{i,j}^{min} = \min_{1 \le u \le n_i} \min_{1 \le v \le n_j} d_{i,j}^{u,v} \tag{6.6}$$

It is easy to see that for any edge connecting t_i and t_j, if $cost'_{i,j}$ is small, then no matter what the final coloring solutions for t_i and t_j are, the cost of this edge in the solution graph is similar. Therefore, we use maximum spanning tree to replace the original solution graph. Note that, $cost'_{i,j}$ is only used during generating maximum spanning tree rather than the following dynamic programming.

6.4.4.2 Dynamic programming solution

After maximum spanning tree is generated, dynamic programming could be applied to find an initial coloring solution. We use the node which has maximal out-degree as the root to generate the tree topology. Then bottom-up method is adopted to construct optimal solutions in the tree. For any node t_i, we maintain a vector $Best[i]$. The entry $Best[i][j]$ stores the best cost over all possible coloring solutions for the sub-tree rooted at node t_i if t_i is choosing coloring solution p_i^j. Suppose it has m children $(x_1, x_2, \cdots, x_m)$, and the vectors for these m children have already been constructed. The vector for t_i can be constructed by the following Formula 6.7. The final total cost is the minimal element of $Best[i]$ if t_i is the root of the tree.

$$Best[i][j] = cost_i[p_i^j] + \sum_{1 \le p \le m} \min_{1 \le z \le n_{x_p}} \left(Best[x_p][z] + cost_{i,x_p}[p_i^j, p_{x_p}^z] \right) \tag{6.7}$$

6.4.5 LP-based refinement

The LP-based refinement technique is presented in Algorithm 17. The idea is that we enumerate all the coloring solutions for one standard cell type while others are fixated. The node whose associated edges' costs are larger is given a higher priority. In Line 4 of Algorithm 17, once the coloring solutions for all the cells are fixed, it is easy to see that minimal cell displacement can be achieved by solving the following LP, where $d_{c_i^{j-1}, c_i^j}$ is the minimal distance to eliminate conflict for adjacent cells c_i^{j-1} and c_i^j in the i-th row.

Minimize: $\sum_{i=1}^{n}\sum_{j=1}^{r_i} q_i^j$

S.T:

$$x_i^j - x_i^{j-1} \geq d_{c_i^{j-1},c_i^j}, \quad \forall 1 \leq i \leq n \wedge 2 \leq j \leq r_i$$
$$x_i^j - o_i^j \leq q_i^j, \quad \forall 1 \leq i \leq n \wedge 1 \leq j \leq r_i$$
$$o_i^j - x_i^j \geq q_i^j, \quad \forall 1 \leq i \leq n \wedge 1 \leq j \leq r_i$$
$$x_i^j \geq \frac{w_{c_i^j}}{2}, \quad \forall 1 \leq i \leq n \wedge 1 \leq j \leq r_i$$
$$x_i^j \leq W - \frac{w_{c_i^j}}{2}, \quad \forall 1 \leq i \leq n \wedge 1 \leq j \leq r_i$$

Algorithm 16 LP-based refinement

```
Calculate the associated edges' costs of each node;
for each node in descending order of associated edges' costs do
    for each coloring solution for this node do
        Minimize the total cell displacement by solving the LP in Section 6.4.5;
        if the value of cost function is better than the current best then
            Update the current best;
            Update the coloring solution for this node.
        end if
    end for
end for
```

6.5 Experimental Results

Our approach is implemented in C++ on a Linux server with Intel Xeon X5550 2.67GHz CPU, 94GB main memory. The benchmarks are derived from [154]'s. Gurobi [161] is used to solve MILP and LP. Since the problem is NP-complete and it cannot be expected to get the optimal solutions for some benchmarks within limited CPU runtime. We limit the MILP solver to run 7200s and report the best solutions within the time limit of MILP solver.

The experimental results are shown in Table 6.1. Compare with MILP solutions, our heuristic approach achieves the same number of stitches. For total cell displacement, the heuristic method is only 2.9% worse than that of MILP solutions on average. However, the heuristic method gets 207× speed up on average. Besides, our method only increases wirelength by less 1% over the initial detailed placement.

Table 6.1 Experiment results: MILP V.S. Heuristic

benchmark	MILP				Heuristic				
	displacement	# of conflicts	# of stitches	runtime(s)	displacement	# of conflicts	# of stitches	WL increase	runtime(s)
alu-70	2.88E+05	0	610	1245	2.94E+05	0	610	0.6%	12
alu-80	6.76E+05	0	610	7200	6.87E+05	0	610	1.4%	14
alu-90	1.94E+06	0	610	7200	1.97E+06	0	610	4.0%	15
byp-70	1.04E+05	0	1134	739	1.04E+05	0	1134	0.0%	21
byp-80	3.85E+05	0	1134	7200	3.68E+05	0	1134	0.1%	28
byp-90	1.54E+06	0	1134	7200	1.60E+06	0	1134	0.7%	31
div-70	1.60E+05	0	1316	3042	1.60E+05	0	1316	0.1%	28
div-80	3.53E+05	0	1316	7200	3.64E+05	0	1316	1.7%	35
div-90	3.62E+06	0	1316	7200	3.61E+06	0	1316	3.8%	32
ecc-70	2.76E+04	0	258	13	2.90E+04	0	258	0.0%	4
ecc-80	8.91E+04	0	258	11	1.09E+05	0	258	0.1%	5
ecc-90	3.55E+05	0	258	23	3.55E+05	0	258	0.9%	6
efc-70	2.84E+04	0	671	420	3.15E+04	0	671	0.0%	6
efc-80	1.14E+05	0	671	4127	1.16E+05	0	671	0.3%	8
efc-90	5.95E+05	0	671	4800	6.00E+05	0	671	2.4%	8
ctl-70	4.55E+04	0	275	351	4.89E+04	0	275	0.0%	10
ctl-80	1.38E+05	0	275	4345	1.40E+05	0	275	0.0%	12
ctl-90	3.49E+05	0	275	7200	3.50E+05	0	275	0.6%	13
top-70	4.95E+05	0	4731	3165	5.12E+05	0	4731	0.0%	326
top-80	1.48E+06	0	4731	7200	1.51E+06	0	4731	0.2%	391
top-90	7.36E+05	0	4731	7200	7.19E+05	0	4731	0.1%	482
Norm.	0.971	1	1	207	1.000	1	1	0.8%	1

6.6 Conclusions

In this chapter, we are focusing on displacement-driven TPL optimization in detailed placement stage under coloring constraints. We recognize this problem as NP-complete, then propose two solutions. The first one is MILP, the other is heuristic approach. We show that the heuristic approach is very efficient compared with MILP by experiment. The proposed heuristic method can produce competitive solution quality within very limited CPU runtime.

CHAPTER 7. Conclusions

Placement is one of the most fundamental problems in VLSI physical design. Although it has been investigated for several decades, it is still hot and challenging mainly because of the continuous increase of design scale. To catch up with this rising tendency of design scale, in this dissertation, we propose several high performance algorithms both in global placement and detailed placement.

In global placement, we propose a very efficient core engine–POLAR, which aims at optimizing wirelength purely. POLAR adopts the rough legalization (look-ahead legalization) idea, and the main contribution is a more efficient algorithm to perform rough legalization. Compared with other academic placers with rough legalization, POLAR achieves the best result on average over academic benchmarks while it has been fastest of all so far. To handle with routing congestion, we extend POLAR by applying the following two ideas. The first idea is routability-driven rough legalization, which is used to migrate congestion globally. The second one is history-based cell inflation, which is used to eliminate very most congested hotspot. This routability-driven placer is called POLAR 2.0. POLAR 2.0 is simple and efficient, it achieves very competitive solution quality while only 2× slower than POLAR. Last but no least, to leverage multi-core system, we explore parallelism in POLAR and propose an ultrafast global placer–POLAR 3.0. We show that parallelizing placement is a challenging problem rather than simple problem. Partitioning used to be considered as out-of-date for large scale placement problem. However, to achieve high scalability, we demonstrate that partitioning is still useful and can be applied within state-of-the-art quadratic placer to trade-off runtime and solution quality. The idea of POLAR 3.0 is to divide global placement iterations into a series of *frames*. In each *frame*, it begins with a roughly legalized placement. Placement-driven partitioning is applied based on the current locations of cells, and then the placement of each partition can be

performed simultaneously. To prevent cells from being restricted in the same partition, varied partitioning scheme is proposed in the *frame*. Experimental results over academic benchmarks show that POLAR 3.0 is almost one order of magnitude faster than the existing academic placers while still can achieve competitive solution quality.

In detailed placement, we consider the impact of triple patterning lithography (TPL) in detailed placement stage. We propose two efficient algorithms for TPL-aware detailed placement with/without coloring constraints. For the problem without coloring constraints, we propose a TPL generation and decomposition method that can leverage existing detailed placement tool as black-box and minimize TPL conflicts and stitch count while maintaining the quality of initial placement maximally. For the problem with coloring constraints, we prove that this problem is NP-hard and give an integer linear programming formulation (ILP). Since solving ILP is time consuming, an efficient heuristic algorithm is proposed to replace ILP to get competitive solution within reasonable runtime.

BIBLIOGRAPHY

[1] Igor L. Markov, Jin Hu, and Myung-Chul Kim. Progress and challenges in VLSI placement research. ICCAD '12, pages 275–282, 2012.

[2] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Journal*, 49:291–307, 1972.

[3] Chung-Kuan Cheng and Ernest S. Kuh. Module placement based on resistive networt optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 3(3):218–225, 1984.

[4] Ren-Song Tsay, Ernest S. Kuh, and Chi-Ping Hsu. Proud: A fast sea-of-gates placement algorithm. DAC '88, pages 318–323, 1988.

[5] G. J. Wipfler, M. Wiesel, and D. A. Mlynski. A combined force and cut algorithm for hierarchical VLSI layout. DAC '82, pages 671–677, 1982.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[7] Wern-Jieh Sun and Carl Sechen. Efficient and effective placement for very large circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 14(3):349–359, 1995.

[8] Maogang Wang, Xiaojian Yang, and Majid Sarrafzadeh. DRAGON2000: Standart-Cell placement tool for large industry curcuits. In *ICCAD '00*, pages 260–263, 2000.

[9] Ameya R. Agnihotri, Satoshi Ono, Chen Li, Mehmet Can Yildiz, Ateen Khatkhate, Cheng-Kok Koh, and Patrick H. Madden. Mixed block placement via fractional cut

recursive bisection. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(5), 2005.

[10] Jarrod A. Roy, David A. Papa, Saurabh N. Adya, Hayward H. Chan, Aaron N. Ng, James F. Lu, and Igor L. Markov. Capo: robust and scalable open-source min-cut floorplacer. In *ISPD '05*, pages 224–226, 2005.

[11] Jürgen M. Kleinhans, Georg Sigl, Frank M. Johannes, and Kurt Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 10(3):356–365, 1991.

[12] Ameya R. Agnihotri, Mehmet Can Yildiz, Ateen Khatkhate, Ajita Mathur, Satoshi Ono, and Patrick H. Madden. Fractional cut: Improved recursive bisection placement. In *ICCAD '03*, pages 307–310, 2003.

[13] Andrew B. Kahng and Sherief Reda. Placement feedback: A concept and method for better min-cut placements. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 357–362, 2004.

[14] Tung-Chieh Chen, Tien-Chang Hsu, Zhe-Wei Jiang, and Yao-Wen Chang. NTUplace: A ratio partitioning based placement algorithm for large-scale mixed-size designs. ISPD '05, pages 236–238, 2005.

[15] Ateen Khatkhate, Chen Li, Ameya R. Agnihotri, Mehmet C. Yildiz, Satoshi Ono, Cheng-Kok Koh, and Patrick H. Madden. Recursive bisection based mixed block placement. ISPD '04, pages 84–89, 2004.

[16] Charles J. Alpert, Gi-Joon Nam, and Paul G. Villarrubia. Effective free space management for cut-based placement via analytical constraint generation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(10):1343–1353, 2003.

[17] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC '82*, pages 175–181, 1982.

[18] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *DAC '97*, pages 526–529, 1997.

[19] Charles J. Alpert and Andrew B. Kahng. Recent directions in netlist partitioning: A survey. *Integr. VLSI J.*, 19(1-2):1–81, August 1995.

[20] T. Bui, C. Heigham, C. Jones, and T. Leighton. Improving the performance of the kernighan-lin and simulated annealing graph bisection algorithms. DAC '89, pages 775–778, 1989.

[21] Jason Cong and M'Lissa Smith. A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design. DAC '93, pages 755–760, 1993.

[22] N. Viswanathan, Min Pan, and C. Chu. FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control. ASP-DAC '07, pages 135–140, 2007.

[23] Natarajan Viswanathan, Gi-Joon Nam, Charles J. Alpert, Paul Villarrubia, Haoxing Ren, and Chris Chu. RQL: Global placement via relaxed quadratic spreading and linearization. DAC '07, pages 453–458, 2007.

[24] Jason Cong, Guojie Luo, Kalliopi Tsota, and Bingjun Xiao. Optimizing routability in large-scale mixed-size placement. ASP-DAC '13, 2013.

[25] Tao Luo and David Z. Pan. DPlace2.0: a stable and efficient analytical placement based on diffusion. ASP-DAC '08, pages 346–351, 2008.

[26] Tung-Chieh Chen, Zhe-Wei Jiang, Tien-Chang Hsu, Hsin-Chen Chen, and Yao-Wen Chang. NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 27(7):1228–1240, 2008.

[27] Myung-Chul Kim, Dong-Jin Lee, and Igor L. Markov. SimPL: an effective placement algorithm. ICCAD '10, pages 649–656, 2010.

[28] Myung-Chul Kim and Igor L. Markov. ComPLx: A competitive primal-dual lagrange optimization for global placement. DAC '12, pages 747–752, 2012.

[29] Andrew B. Kahng, Sherief Reda, and Qinke Wang. Aplace: A general analytic placement framework. ISPD '05, pages 233–235, 2005.

[30] Ulrich Brenner, Anna Hermann, Nils Hoppmann, and Philipp Ochsendorf. BonnPlace: A self-stabilizing placement framework. ISPD '15, pages 9–16, 2015.

[31] P. Spindler, U. Schlichtmann, and F. M. Johannes. Kraftwerk2: A fast force-directed quadratic placement approach using an accurate net model. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 27(8):1398–1411, 2008.

[32] Wenxing Zhu, Jianli Chen, Zheng Peng, and Genghua Fan. Nonsmooth optimization method for VLSI global placement. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 34(4):642–655, 2015.

[33] Tao Lin, Chris Chu, Joseph R. Shinnerl, Ismail Bustany, and Ivailo Nedelchev. POLAR: Placement based on novel rough legalization and refinement. ICCAD '13, pages 357–362, 2013.

[34] Jingwei Lu, Pengwen Chen, Chin-Chih Chang, Lu Sha, Dennis J-.H. Huang, Chin-Chi Teng, and Chung-Kuan Cheng. ePlace: Electrostatics based placement using nesterov's method. DAC '14, pages 121:1–121:6, 2014.

[35] Jackey Z. Yan, Natarajan Viswanathan, and Chris Chu. Handling complexities in modern large-scale mixed-size placement. DAC '09, pages 436–441, 2009.

[36] Jackey Z. Yan, Natarajan Viswanathan, and Chris Chu. An effective floorplan-guided placement algorithm for large-scale mixed-size designs. *ACM Trans. Des. Autom. Electron. Syst.*, 19(3):29:1–29:25, June 2014.

[37] Meng-Kai Hsu and Yao-Wen Chang. Unified analytical global placement for large-scale mixed-size circuit designs. ICCAD '10, pages 657–662, 2010.

[38] Charles J. Alpert, Zhuo Li, Michael D. Moffitt, Gi-Joon Nam, Jarrod A. Roy, and Gustavo Tellez. What makes a design difficult to route. ISPD '10, pages 7–12, 2010.

[39] Ulrich Brenner and André Rohe. An effective congestion driven placement framework. ISPD '02, pages 6–11, 2002.

[40] Jason Cong, Guojie Luo, Kalliopi Tsota, and Bingjun Xiao. Optimizing routability in large-scale mixed-size placement. In *ASP-DAC '13*, pages 441–446, 2013.

[41] Xu He, Wing-Kai Chow, and Evangeline F.Y. Young. SRP: Simultaneous routing and placement for congestion refinement. In *Proceedings of the 2013 ACM International Symposium on International Symposium on Physical Design*, ISPD '13, pages 108–113, 2013.

[42] Xu He, Tao Huang, Linfu Xiao, Haitong Tian, Guxin Cui, and Evangeline F. Y. Young. Ripple: an effective routability-driven placer by iterative cell movement. ICCAD '11, pages 74–79, 2011.

[43] Xu He, Tao Huang, Wing-Kai Chow, Jian Kuang, Ka-Chun Lam, Wenzan Cai, and Evangeline F. Y. Young. Ripple 2.0: high quality routability-driven placement via global router integration. DAC '13, pages 152:1–152:6, 2013.

[44] Myung-Chul Kim, Jin Hu, Dong-Jin Lee, and Igor L. Markov. A SimPLR method for routability-driven placement. ICCAD '11, pages 67–73, 2011.

[45] Tao Lin and Chris Chu. POLAR 2.0: An effective routability-driven placer. DAC '14, pages 123:1–123:6, 2014.

[46] Meng-Kai Hsu, Yi-Fang Chen, Chau-Chin Huang, Tung-Chieh Chen, and Yao-Wen Chang. Routability-driven placement for hierarchical mixed-size circuit designs. DAC '13, pages 151:1–151:6, 2013.

[47] Wenting Hou, Hong Yu, Xianlong Hong, Yici Cai, Weimin Wu, Jun Gu, and William H. Kao. A new congestion-driven placement algorithm based on cell inflation. In *Proceedings*

of the 2001 Asia and South Pacific Design Automation Conference, ASP-DAC '01, pages 605–608, 2001.

[48] Min Pan and Chris Chu. IPR: An integrated placement and routing algorithm. DAC '07, pages 59–62, 2007.

[49] Jarrod A. Roy, Natarajan Viswanathan, Gi-Joon Nam, Charles J. Alpert, and Igor L. Markov. CRISP: Congestion reduction by iterated spreading during placement. ICCAD '09, pages 357–362, 2009.

[50] Yaoguang Wei, Cliff Sze, Natarajan Viswanathan, Zhuo Li, Charles J. Alpert, Lakshmi Reddy, Andrew D. Huber, Gustavo E. Tellez, Douglas Keller, and Sachin S. Sapatnekar. GLARE: Global and local wiring aware routability evaluation. DAC '12, pages 768–773, 2012.

[51] Natarajan Viswanathan, Charles Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. ICCAD-2012 CAD contest in design hierarchy aware routability-driven placement and benchmark suite. ICCAD '12, pages 345–348, 2012.

[52] Natarajan Viswanathan, Charles J. Alpert, Cliff Sze, Zhuo Li, Gi-Joon Nam, and Jarrod A. Roy. The ISPD-2011 routability-driven placement contest and benchmark suite. ISPD '11, pages 141–146, 2011.

[53] Vladimir Yutsis, Ismail S. Bustany, David Chinnery, Joseph R. Shinnerl, and Wen-Hao Liu. ISPD 2014 benchmarks with sub-45nm technology rules for detailed-routing-driven placement. ISPD '14, pages 161–168, 2014.

[54] Karthik Rajagopal, Tal Shaked, Yegna Parasuram, Tung Cao, Amit Chowdhary, and Bill Halpin. Timing driven force directed placement with physical net constraints. ISPD '03, pages 60–66, 2003.

[55] William Swartz and Carl Sechen. Timing driven placement for large standard cell circuits. DAC '95, pages 211–215, 1995.

[56] Yih-Chih Chou and Youn-Long Lin. A performance-driven standard-cell placer based on a modified force-directed algorithm. ISPD '01, pages 24–29, 2001.

[57] Bill Halpin, C. Y. Roger Chen, and Naresh Sehgal. Timing driven placement using physical net constraints. DAC '01, pages 780–783, 2001.

[58] Ren-Song Tsay and Juergen Koehl. An analytic net weighting approach for performance optimization in circuit placement. DAC '91, pages 620–625, 1991.

[59] Wonjoon Choi and Kia Bazargan. Incremental placement for timing optimization. ICCAD '03, pages 463–466, 2003.

[60] Amit Chowdhary, Karthik Rajagopal, Satish Venkatesan, Tung Cao, Vladimir Tiourin, Yegna Parasuram, and Bill Halpin. How accurately can we model timing in a placement engine? DAC '05, pages 801–806, 2005.

[61] Chanseok Hwang and Massoud Pedram. Timing-driven placement based on monotone cell ordering constraints. ASP-DAC '06, pages 201–206, 2006.

[62] Qingzhou (Ben) Wang, John Lillis, and Shubhankar Sanyal. An LP-based methodology for improved timing-driven placement. ASP-DAC '05, pages 1139–1143, 2005.

[63] Natarajan Viswanathan, Gi-Joon Nam, Jarrod A. Roy, Zhuo Li, Charles J. Alpert, Shyam Ramji, and Chris Chu. ITOP: Integrating timing optimization within placement. ISPD '10, pages 83–90, 2010.

[64] A. B. Kahng and Q. Wang. An analytic placer for mixed-size placement and timing-driven placement. ICCAD '04, pages 565–572, 2004.

[65] Ashutosh Chakraborty and David Z. Pan. PASAP: Power aware structured asic placement. ISLPED '10, pages 395–400, 2010.

[66] Yongseok Cheon, Pei-Hsin Ho, Andrew B. Kahng, Sherief Reda, and Qinke Wang. Power-aware placement. DAC '05, pages 795–800, 2005.

[67] Bin Liu, Yici Cai, Qiang Zhou, and Xianlong Hong. Power driven placement with layout aware supply voltage assignment for voltage island generation in dual-vdd designs. ASP-DAC '06, pages 582–587, 2006.

[68] Andrew B. Kahng, Bao Liu, and Qinke Wang. Stochastic power/ground supply voltage prediction and optimization via analytical placement. *IEEE Trans. Very Large Scale Integr. Syst.*, 15(8):904–912, aug 2007.

[69] Andrew B. Kahng, Bao Liu, and Qinke Wang. Supply voltage degradation aware analytical placement. ICCD '05, pages 437–443, 2005.

[70] Yao-Tsung Chang, Chih-Cheng Hsu, Mark Po-Hung Lin, Yu-Wen Tsai, and Sheng-Fong Chen. Post-placement power optimization with multi-bit flip-flops. ICCAD '10, pages 218–223, 2010.

[71] Guoqiang Chen and Sachin Sapatnekar. Partition-driven standard cell thermal placement. ISPD '03, pages 75–80, 2003.

[72] Bernd Obermeier and Frank M. Johannes. Temperature-aware global placement. ASP-DAC '04, pages 143–148, 2004.

[73] Craig Beebe, Jo Dale Carothers, and Alfonso Ortega. MCM placement using a realistic thermal model. GLSVLSI '00, pages 189–192, 2000.

[74] Ching-Han Tsai and Sung-Mo (Steve) Kang. Standard cell placement for even on-chip thermal distribution. ISPD '99, pages 179–184, 1999.

[75] Po-Hung Lin, Hongbo Zhang, Martin D. F. Wong, and Yao-Wen Chang. Thermal-driven analog placement considering device matching. DAC '09, pages 593–598, 2009.

[76] Wilm Donath, Prabhakar Kudva, Leon Stok, Lakshmi Reddy, Andrew Sullivan, Kanad Chakraborty, and Paul Villarrubia. Transformational placement and synthesis. DATE '00, pages 194–201, 2000.

[77] Haoxing Ren, David Z. Pan, and David S. Kung. Sensitivity guided net weighting for placement driven synthesis. ISPD '04, pages 10–17, 2004.

[78] Jinan Lou, Amir H. Salek, and Massoud Pedram. An exact solution to simultaneous technology mapping and linear placement problem. ICCAD '97, pages 671–675, 1997.

[79] Majid Sarrafzadeh, David Knol, and Gustavo Tellez. Unification of budgeting and placement. DAC '97, pages 758–761, 1997.

[80] Kai-hui Chang, Igor L. Markov, and Valeria Bertacco. SafeResynth: A new technique for physical synthesis. *Integr. VLSI J.*, 41(4):544–556, jul 2008.

[81] Yifang Liu, Rupesh S. Shelar, and Jiang Hu. Delay-optimal simultaneous technology mapping and placement with applications to timing optimization. ICCAD '08, pages 101–106, 2008.

[82] Yanbin Jiang and Sachin S. Sapatnekar. An integrated algorithm for combined placement and libraryless technology mapping. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '99, pages 102–106, 1999.

[83] Amir H. Salek, Jinan Lou, and Massoud Pedram. A DSM design flow: Putting floorplanning, technology-mapping, and gate-placement together. DAC '98, pages 128–134, 1998.

[84] Wilsin Gosti, Sunil P. Khatri, and Alberto L. Sangiovanni-Vincentelli. Addressing the timing closure problem by integrating logic optimization and placement. ICCAD '01, pages 224–231, 2001.

[85] Xing Wei, Wai-Chung Tang, Yu-Liang Wu, Cliff Sze, and Charles Alpert. WRIP: Logic restructuring techniques for wirelength-driven incremental placement. GLSVLSI '12, pages 327–332, 2012.

[86] Kuan-Hsien Ho, Yen-Pin Chen, Jia-Wei Fang, and Yao-Wen Chang. Eco timing optimization using spare cells and technology remapping. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29(5):697–710, may 2010.

[87] Shyam Ramji and Nagu R. Dhanwada. Design topology aware physical metrics for placement analysis. GLSVLSI '03, pages 186–191, 2003.

[88] Ulrich Brenner. VLSI legalization with minimum perturbation by iterative augmentation. DATE '12, pages 1385–1390, 2012.

[89] Ulrich Brenner, Anna Pauli, and Jens Vygen. Almost optimum placement legalization by minimum cost flow and dynamic programming. ISPD '04, pages 2–9, 2004.

[90] Minsik Cho, Haoxing Ren, Hua Xiang, and Ruchir Puri. History-based VLSI legalization using network flow. DAC '10, pages 286–291, 2010.

[91] Yu-Min Lee, Tsung-You Wu, and Po-Yi Chiang. A hierarchical bin-based legalizer for standard-cell designs with minimal disturbance. ASP-DAC '10, pages 568–573, 2010.

[92] Andrew B. Kahng, Igor L. Markov, and Sherief Reda. On legalization of row-based placements. GLSVLSI '04, pages 214–219, 2004.

[93] Peter Spindler, Ulf Schlichtmann, and Frank M. Johannes. Abacus: Fast legalization of standard cell circuits with minimal movement. ISPD '08, pages 47–53, 2008.

[94] A. E. Caldwell, A. B. Kahng, and B. I. L. Markov. Optimal partitioners and end-case placers for standard-cell layout. In *ISPD '99*, pages 90–96.

[95] Ulrich Brenner and Jens Vygen. Faster optimal single-row placement with fixed ordering. In *DATE '00*, pages 117–121, 2000.

[96] Wing-Kai Chow, Jian Kuang, Xu He, Wenzan Cai, and Evangeline F.Y. Young. Cell density-driven detailed placement with displacement constraint. ISPD '14, pages 3–10, 2014.

[97] Sergiy Popovych, Hung-Hao Lai, Chieh-Min Wang, Yih-Lang Li, Wen-Hao Liu, and Ting-Chi Wang. Density-aware detailed placement with instant legalization. DAC '14, pages 122:1–122:6, 2014.

[98] Lijuan Luo, Qiang Zhou, Xianlong Hong, and Hanbin Zhou. Multi-stage detailed placement algorithm for large-scale mixed-mode layout design. ICCSA '05, pages 896–905, 2005.

[99] Shuai Li and Cheng-Kok Koh. Mixed integer programming models for detailed placement. ISPD '12, pages 87–94, 2012.

[100] Shuai Li and Cheng-kok Koh. MIP-based detailed placer for mixed-size circuits. ISPD '14, pages 11–18, 2014.

[101] Satoshi Ono and Patrick H. Madden. On structure and suboptimality in placement. ASP-DAC '05, pages 331–336, 2005.

[102] Andrew B. Kahng, Paul Tucker, and Alexander Zelikovsky. Optimization of linear placements for wirelength minimization with free sites. In *ASP-DAC '99*, pages 241–244, 1999.

[103] Min Pan, N. Viswanathan, and C. Chu. An efficient and effective detailed placement algorithm. ICCAD '05, pages 48–55, 2005.

[104] Brent Goplen and Sachin Sapatnekar. Thermal via placement in 3D ICs. ISPD '05, pages 167–174, 2005.

[105] Jason Cong, Guojie Luo, and Yiyu Shi. Thermal-aware cell and through-silicon-via co-placement for 3D ICs. DAC '11, pages 670–675, 2011.

[106] Brent Goplen and Sachin Spatnekar. Placement of 3D ICs with thermal and interlayer via considerations. DAC '07, pages 626–631, 2007.

[107] Krit Athikulwongse, Mohit Pathak, and Sung Kyu Lim. Exploiting die-to-die thermal coupling in 3D IC placement. DAC '12, pages 741–746, 2012.

[108] Brent Goplen and Sachin Sapatnekar. Efficient thermal placement of standard cells in 3D ICs using a force directed approach. ICCAD '03, pages 86–89, 2003.

[109] Karthik Balakrishnan, Vidit Nanda, Siddharth Easwar, and Sung Kyu Lim. Wire congestion and thermal aware 3D global placement. ASP-DAC '05, pages 1131–1134, 2005.

[110] Charles Alpert, Zhuo Li, Gi-Joon Nam, C. N. Sze, Natarajan Viswanathan, and Samuel I. Ward. Placement: Hot or not? ICCAD '12, pages 283–290, 2012.

[111] Chris Chu. *Electronic Design Automation-synthesis, verification and test*, chapter placement. Morgan Kaufmann, 2009.

[112] Jens Vygen. Algorithms for large-scale flat placement, 1997.

[113] Hans Eisenmann and Frank M. Johannes. Generic global placement and floorplanning. DAC '98, pages 269–274, 1998.

[114] Fan Mo, Abdallah Tabbara, and Robert K. Brayton. A force-directed macro-cell placer. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '00, pages 177–181, 2000.

[115] Natarajan Viswanathan and Chris Chong-Nuen Chu. Fastplace: Efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model. ISPD '04, pages 26–33, 2004.

[116] Bo Hu, Yue Zeng, and Malgorzata Marek-Sadowska. mFAR: Fixed-points-addition-based VLSI placement algorithm. ISPD '05, pages 239–241, 2005.

[117] Myung-Chul Kim, Natarajan Viswanathan, Charles J. Alpert, Igor L. Markov, and Shyam Ramji. MAPLE: Multilevel adaptive placement for mixed-size designs. ISPD '12, pages 193–200, 2012.

[118] W.C. Naylor, R. Donelly, and L. Sha. Non-linear optimization system and method for wire length and delay optimization for an automatic electric circuit placer, 2001. US Patent 6,301,693.

[119] David G. Luenberger. *Linear and nonlinear programming.* Kluwer Academic Publ., Boston, Dordrecht, London, 2003.

[120] Kenneth Joseph Arrow, Leonid Hurwicz, and Hirofumi Uzawa. *Studies in linear and non-linear programming.* Stanford mathematical studies in the social sciences. Stanford university press, Stanford,Ca, 1972.

[121] Angelia Nedic and Dimitri P. Bertsekas. Incremental subgradient methods for nondifferentiable optimization. *SIAM Journal on Optimization*, 12(1):109–138, 2001.

[122] Charles J. Alpert, Andrew B. Kahng, Gi-Joon Nam, Sherief Reda, and Paul Villarrubia. A semi-persistent clustering technique for VLSI circuit placement. In *ISPD '05*, pages 200–207, 2005.

[123] Jackey Z. Yan, Chris Chu, and Wai-Kei Mak. SafeChoice: a novel clustering algorithm for wirelength-driven placement. ISPD '10, pages 185–192, 2012.

[124] Gi-Joon Nam, Charles J. Alpert, Paul Villarrubia, Bruce Winter, and Mehmet Yildiz. The ISPD2005 placement contest and benchmark suite. ISPD '05, pages 216–220, 2005.

[125] Gi-Joon Nam. ISPD 2006 placement contest: Benchmark suite and results. ISPD '06, pages 167–167, 2006.

[126] Tao Lin and Chris Chu. Tpl-aware displacement-driven detailed placement refinement with coloring constraints. ISPD '15, pages 75–80, 2015.

[127] Gi-Joon Nam, S. Reda, C. J. Alpert, P. G. Villarrubia, and A. B. Kahng. A fast hierarchical quadratic placement algorithm. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 25(4):678–691, 2006.

[128] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2003.

[129] Chen Li, Min Xie, Cheng-Kok Koh, J. Cong, and P. H. Madden. Routability-driven placement and white space allocation. ICCAD '04, pages 394–401.

[130] Saurabh N. Adya and Igor L. Markov. Consistent placement of macro-blocks using floorplanning and standard-cell placement. ISPD '02, pages 12–17.

[131] Charles J. Alpert, Zhuo Li, Michael D. Moffitt, Gi-Joon Nam, Jarrod A. Roy, and Gustavo Tellez. What makes a design difficult to route. ISPD '10, pages 7–12, 2010.

[132] Jarrod A. Roy, James F. Lu, and Igor L. Markov. Seeing the forest and the trees: Steiner wirelength optimization in placemen. ISPD '06, pages 78–85, 2006.

[133] Jin Hu, Myung-Chul Kim, and Igor L. Markov. Taming the complexity of coordinated place and route. DAC '13, pages 150:1–150:7, 2013.

[134] Meng-Kai Hsu, Sheng Chou, Tzu-Hen Lin, and Yao-Wen Chang. Routability-driven analytical placement for mixed-size circuit designs. ICCAD '11, pages 80–84, 2011.

[135] Wen-Hao Liu, Cheng-Kok Koh, and Yih-Lang Li. Optimization of placement solutions for routability. DAC '13, pages 153:1–153:9, 2013.

[136] Yue Xu, Yanheng Zhang, and Chris Chu. FastRoute 4.0: global router with efficient via minimization. ASP-DAC '09, pages 576–581, 2009.

[137] Jin Hu, Jarrod A. Roy, and Igor L. Markov. Completing high-quality global routes. ISPD '10, pages 35–41, 2010.

[138] Peter Spindler and Frank M. Johannes. Fast and accurate routing demand estimation for efficient routability-driven placement. DATE '07, pages 1226–1231, 2007.

[139] Ulrich Brenner and André Rohe. An effective congestion driven placement framework. In *ISPD '02*, pages 6–11, 2002.

[140] Yanheng Zhang and Chris Chu. CROP: Fast and effective congestion refinement of placement. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 344–350, 2009.

[141] Larry McMurchie and Carl Ebeling. PathFinder: A negotiation-based performance-driven router for fpgas. FPGA '95, pages 111–117, 1995.

[142] Wen-Hao Liu, Wei-Chun Kao, Yih-Lang Li, and Kai-Yuan Chao. Multi-threaded collision-aware global routing with bounded-length maze routing. DAC '10, pages 200–205, 2010.

[143] Myung-Chul Kim, Dongjin Lee, and Igor L. Markov. SimPL: An effective placement algorithm. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(1):50–60, 2012.

[144] Intel. Intel Math Kernel Library. https://software.intel.com.

[145] Ismail S. Bustany, David Chinnery, Joseph R. Shinnerl, and Vladimir Yutsis. ISPD 2015 benchmarks with fence regions and routing blockages for detailed-routing-driven placement. ISPD '15, pages 157–164, 2015.

[146] Bei Yu, Kun Yuan, Boyang Zhang, Duo Ding, and David Z. Pan. Layout decomposition for triple patterning lithography. ICCAD '11, pages 1–8, 2011.

[147] Shao-Yun Fang, Yao-Wen Chang, and Wei-Yu Chen. A novel layout decomposition algorithm for triple patterning lithography. DAC '12, pages 1185–1190, 2012.

[148] Andrew B. Kahng, Chul-Hong Park, Xu Xu, and Hailong Yao. Layout decomposition for double patterning lithography. ICCAD '08, pages 465–472, 2008.

[149] Jian Kuang and Evangeline F. Y. Young. An efficient layout decomposition approach for triple patterning lithography. DAC '13, pages 69–75, 2013.

[150] Ye Zhang, Wai-Shing Luk, Hai Zhou, Changhao Yan, and Xuan Zeng. Layout decomposition with pairwise coloring for multiple patterning lithography. ICCAD '13, pages 170–177, 2013.

[151] Haitong Tian, Hongbo Zhang, Qiang Ma, Zigang Xiao, and Martin D. F. Wong. A polynomial time triple patterning algorithm for cell based row-structure layout. ICCAD '12, pages 57–64, 2012.

[152] L Liebmann and D Pletromonaco. Decomposition-aware standard cell design flows to enable double-patterning technology. In *Proceedings of SPIE*, volume 7974(1), 2011.

[153] Jhih-Rong Gao, Bei Yu, and David Z. Pan. Self-aligned double patterning friendly configuration for standard cell library considering placement. In *Proceedings of SPIE*, volume 8684(6), 2013.

[154] Bei Yu, Xiaoqing Xu, Jhih-Rong Gao, and David Z. Pan. Methodology for standard cell compliance and detailed placement for triple patterning lithography. ICCAD '13, pages 349–356, 2013.

[155] Haitong Tian, Yuelin Du, Hongbo Zhang, Zigang Xiao, and Martin D. F. Wong. Constrained pattern assignment for standard cell based triple patterning lithography. ICCAD '13, pages 178–185, 2013.

[156] Bei Yu, Yen-Hung Lin, Gerard Luk-Pat, Duo Ding, Kevin Lucas, and David Z. Pan. A high-performance triple patterning layout decomposer with balanced density. In *ICCAD '13*, pages 163–169, 2013.

[157] Zihao Chen, Hailong Yao, and Yici Cai. SUALD: Spacing uniformity-aware layout decomposition in triple patterning lithography. In *ISQED '13*, pages 566–571, 2013.

[158] Qiang Ma, Hongbo Zhang, and Martin D. F. Wong. Triple patterning aware routing and its comparison with double patterning aware routing in 14nm technology. DAC '12, pages 591–596, 2012.

[159] Yen-Hung Lin, Bei Yu, David Z. Pan, and Yih-Lang Li. TRIAD: A triple patterning lithography aware detailed router. ICCAD '12, pages 123–129, 2012.

[160] Michael R Garey and David S Johnson. *A Guide to the Theory of NP-Completeness*. Macmillan Higher Education, 1979.

[161] Gurobi. http://www.gurobi.com.